Bachelor Thesis

September 9, 2018

"*What Should I Change in My Patch?*"

An Empirical Investigation of Relevant Changes and Automation Needs in Modern Code Review

Nik Zaugg

of Bern, Schweiz (12-716-734)

supervised by

Prof. Dr. Harald C. Gall

Dr. Sebastiano Panichella

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Bachelor Thesis

"*What Should I Change in My Patch?*"

An Empirical Investigation of Relevant Changes and Automation Needs in Modern Code Review

Nik Zaugg

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor Thesis

Author: Nik Zaugg, nik.zaugg@uzh.ch
URL: http://www.ifi.uzh.ch/en/seal.html
Project period: 13.03.2018 - 13.09.2018

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

First of all, I would like to express my gratitude to Dr. Sebastiano Panichella for supporting me throughout the six months of this research. Whenever I faced difficulties, he was there to guide me in the right direction and give constructive comments and encouragement. It was a pleasure working on this interesting topic with him.

Furthermore, my thanks go to Prof. Dr. Harad Gall for allowing me to write my Bachelor thesis with the Software Evolution and Architecture Lab at the University of Zurich.

A special thanks goes to Noemi, who supported me during these challenging months with her time and patience; and to Alex and Moritz for listening to my ideas and the discussions we had.

Finally, I want to thank my parents for their support and believing in me throughout my studies.

Abstract

Recent research has shown that available tools for Modern Code Review (MCR) are still far from meeting the current expectations of developers. The objective of this thesis is to investigate the most recurrent change types in MCR as well as the developers' expectations and needs regarding the automation of reviewing activities that, from a *developer point of view*, are still needed to facilitate MCR activities by considering current literature, manually analyzing code review changes and conducting a survey. Additionally, we explore approaches and tools that are still needed to facilitate MCR activities and extract various metrics describing a patch in *Gerrit* code review with the goal to predict required changes in the future. To that end, we first empirically elicited a taxonomy of recurrent review *change types* that characterize code reviews. The taxonomy was designed by performing three steps: (i) we generated an initial version of the taxonomy by qualitatively and quantitatively analyzing 211 review commits and 648 review comments of ten open source projects; then (ii) we integrated topics and code review *change types* of an existing taxonomy available from the literature into this initial taxonomy; finally, (iii) we surveyed 52 developers to integrate eventually missing *change types* into the taxonomy. We then evaluated the survey feedback to find out more about current developers' expectations towards code review and how code review activities can be facilitated by novel tools and approaches. Results of our taxonomy evaluation supports previous research, showing that the majority of changes in code review are related to maintainability issues. Furthermore, our findings highlight that the availability of emerging development technologies (*e.g.*, cloud-based technologies) and practices (*e.g.*, continuous delivery and continuous integration) further widens the gap between the expectations developers have towards code review and its outcome. This has pushed developers to perform additional activities during code reviews and shows that additional types of feedback are expected from reviewers, especially regarding changes in non-source-code artifacts (*e.g.*, configurations of Automated Static Analysis Tools). Our survey participants provided recommendations, specified techniques to employ, and highlighted the data to analyze for implementing approaches able to automate the code review activities related to our taxonomy. Most promising recommendations towards the automation of MCR involve the use of Machine Learning and Natural Language Processing techniques to study recurrent patterns and anti-patterns as well as code, change and object-oriented metrics. This study sheds some more light on the most critical and recurring changes in code review, the developers' expectations and needs, and ultimately on the approaches and tools that are still needed to facilitate MCR activities. We believe that this is an essential step towards closing the gap between developers' expectations in code review and its outcome as well as supporting the vision of full or partial automation in MCR.

Zusammenfassung

Jüngste Untersuchungen haben gezeigt, dass die verfügbaren Tools für Modern Code Review (MCR) nicht den aktuellen Erwartungen der Entwickler entsprechen. Das Ziel dieser Arbeit ist es, die am häufigsten auftretenden Change-Typen in MCR sowie die Erwartungen und Bedürfnisse der Entwickler in Bezug auf die Automatisierung von Review-Aktivitäten zu untersuchen, die aus Sicht des Entwicklers immer noch benötigt werden um MCR-Aktivitäten zu erleichtern. Dazu berücksichtigen wir die aktuelle Literatur, analysieren manuell Code Review-Änderungen und führen eine Umfrage durch. Darüber hinaus untersuchen wir Ansätze und Tools, die noch benötigt werden, um MCR-Aktivitäten zu erleichtern und extrahieren verschiedene Metriken, die einen Patch in *Gerrit* Code Review beschreiben, mit dem Ziel, erforderliche Änderungen in der Zukunft vorherzusagen. Dazu haben wir zunächst empirisch eine Taxonomie von wiederkehrenden Review Change-Typen herausgearbeitet. Die Taxonomie wurde in drei Schritten erstellt: (i) wir haben eine erste Version der Taxonomie erstellt, indem wir 211 Review Commits und 648 Review-Kommentare von zehn Open-Source-Projekten qualitativ und quantitativ analysiert haben; dann (ii) haben wir Code Review Change-Typen einer aus der Literatur verfügbaren Taxonomie in diese erste Taxonomie integriert; schliesslich (iii) haben wir 52 Entwickler befragt, um eventuell fehlende Change-Typen in die Taxonomie zu integrieren. Anschliessend werteten wir das Feedback der Umfrage aus, um mehr über die Erwartungen der aktuellen Entwickler an Code Review zu erfahren und herauszufinden, wie Code Review durch neue Tools und Ansätze erleichtert werden kann. Die Ergebnisse unserer Taxonomieauswertung unterstützen frühere Untersuchungen und zeigen, dass die Mehrheit der Änderungen in Code Review mit Problemen der Wartbarkeit zusammenhängt. Darüber hinaus zeigen unsere Ergebnisse, dass die Verfügbarkeit neuer Entwicklungstechnologien (*e.g.*, Cloud-basierte Technologien) und -praktiken (*e.g.*, Continuous Delivery und Continuous Integration) die Lücke zwischen den Erwartungen der Entwickler an Code Review und deren Ergebnis weiter vergrössert. Dies führt dazu, dass zusätzliche Arten von Feedback von den Reviewern erwartet werden, insbesondere in Bezug auf Änderungen an Nicht-Quellcode-Objekten (*e.g.*, Konfigurationen von Automated Static Analysis Tools). Unsere Umfrageteilnehmer gaben Empfehlungen ab, spezifizierten Techniken die anzuwenden sind und hoben die zu analysierenden Daten für die Implementierung von Ansätzen hervor, die in der Lage sind, Change-Typen unserer Taxonomie zu automatisieren. Die vielversprechendsten Empfehlungen zur Automatisierung von MCR beinhalten den Einsatz von Machine Learning und Natural Language Processing Techniken zur Untersuchung wiederkehrender Muster sowie von Code-, Change- und objektorientierten Metriken. Diese Studie wirft mehr Licht auf die kritischsten und wiederkehrenden Änderungen in Code Review, die Erwartungen und Bedürfnisse der Entwickler und auf die Ansätze und Tools, die noch benötigt werden, um MCR-Aktivitäten zu erleichtern. Wir glauben, dass dies ein wesentlicher Schritt ist, um die Lücke zwischen den Erwartungen der Entwickler an Code Review und dessen Esrgebnissen zu schliessen und die Vision einer vollständigen oder teilweisen Automatisierung in MCR zu unterstützen.

Contents

1 Introduction **1**
1.1 Modern Code Review . . . 1
1.2 Motivation & Research Questions . . . 3

2 Related Work **7**
2.1 Modern Code Review Process and Practices . . . 7
2.2 Automation in Modern Code Review . . . 8

3 Methodology **11**
3.1 Manual Classification and Validation . . . 11
3.2 Survey . . . 17
3.3 Proof of Concept . . . 19

4 Results and Discussion **23**
4.1 Change Types in MCR . . . 23
4.1.1 Initial Taxonomies . . . 23
4.1.2 CRAM Model . . . 24
4.1.3 Recurrent and Novel Change Types in MCR . . . 27
4.2 Towards the Automation of MCR . . . 34
4.2.1 Emerging Automation Needs . . . 36
4.3 Enabling Automation: a Proof of Concept . . . 45
4.3.1 Metrics . . . 45
4.3.2 Feature Extraction . . . 49
4.3.3 Change Type Prediction . . . 49
4.4 Threats to Validity . . . 52

5 Conclusion **53**

List of Figures

1.1 Fagan (Traditional) Inspection . . . 1
1.2 General Development Process . . . 2
1.3 *Gerrit* Development Process . . . 3
1.4 Pre-commit and post-commit code reviews. . . . 4

3.1 Overview Research Approach . . . 11
3.2 Example Gerrit (https://git.eclipse.org/r/#/c/44251/) . . . 14
3.3 Example Gerrit side-by-side diff view (https://git.eclipse.org/r/#/c/44251/) . . . 14
3.4 Example Gerrit Log for the M2E Project . . . 15
3.5 Taxonomy of changes of code under review by Beller *et al.* [15] . . . 16

4.1 Approach for Evaluating Developers' Automation Needs . . . 36
4.2 The General Defect Classification (GDC) by Beller *et al.* [16]. . . . 48
4.3 Feature Extraction Process . . . 50

List of Tables

3.1 Characteristics of the analyzed projects. . . . 13
3.2 Survey questions. (MC: Multiple Choice, O: Open answer) . . . 17
3.3 Information About Survey Participants . . . 18
3.4 Encoding of Survey Feedback . . . 20
3.5 Clustering of Survey Feedback . . . 20

4.1 Initial taxonomy of changes in CR . . . 24
4.2 Intermediate taxonomy after validation with the taxonomy by Beller *et al.* [15] - Part 1 25
4.3 Intermediate taxonomy after validation with the taxonomy by Beller *et al.* [15] - Part 2 26
4.4 CRAM (Code Review chAnges-Model) - Part 1 . . . 28
4.5 CRAM (Code Review chAnges-Model) - Part 2 . . . 29
4.6 CRAM (Code Review chAnges-Model) - Part 3 . . . 30
4.7 Distribution of changes in CRAM by *Artifact* . . . 30
4.8 Distribution of changes in CRAM by *Production Code and Test Code* . . . 30
4.9 Distribution of changes in CRAM by *Functionality and Maintainability* . . . 30
4.10 Distribution of changes in CRAM by *Other Changes* . . . 31
4.11 Distribution of changes in CRAM by *Category* . . . 31
4.12 Distribution of changes in CRAM by *top 5 Topics* . . . 31
4.13 Distribution of changes in CRAM by *top 15 Detailed Changes* . . . 32
4.14 Q1.3: Distribution of recurrent change topics . . . 35
4.15 Q1.4: Expected feedback in MCR . . . 35
4.16 Q1.5: Feedback received in MCR . . . 36
4.17 RQ_2: Developers' Envisioned Solutions. . . . 37
4.18 RQ_2: Developers' Recommendations. . . . 40
4.19 RQ_2: Developers' Techniques. . . . 41
4.20 RQ_2: Developers' Data. . . . 42
4.21 Change Metrics . . . 46
4.22 CK-Metrics . . . 47

5.1 Main Findings and Needed Future Work . . . 54

List of Listings

3.1 Example Gerrit Query 15

Chapter 1

Introduction

1.1 Modern Code Review

Code Review. Code Review (CR) is a relevant development approach and widely agreed-on best practice. CR involves manual inspection of source code written by others with the goal of identifying potential defects, bugs and providing an author timely feedback on his or her changes to improve the quality or functionality before the changes are deployed into a live environment. [4, 17, 29, 57]. There exist different approaches to CR which can be subdivided into formal and lightweight code reviews. The former is known as the Fagan (Traditional) Inspection [29] and the latter as Modern Code Review (MCR).

Fagan (Traditional) Inspection. The Formal Inspection process by Fagan is considered the more heavy-weight and formal in comparison to MCR. Characteristics of code inspection include considerable processes similar to waterfall-like procedures. This can include checklists, group meetings, an expert panel and other formal requirements. Although this approach is very thorough and effective, it has the disadvantage that it can take too much time to be considered practical. Furthermore, with current developer teams sometimes being spread around the world and developing in different time zones, it can quickly become difficult to coordinate such reviews [29]. Generally, the Fagan Inspection consists of the steps seen in Figure 1.1.

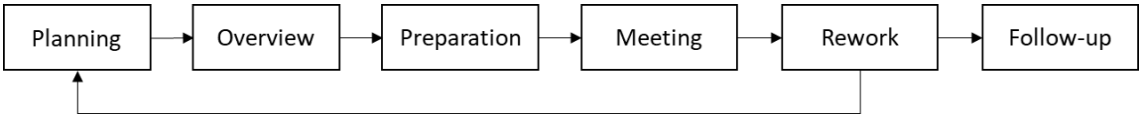

Figure 1.1: Fagan (Traditional) Inspection

The *Planning* phase involves the preparation of materials, arranging of a meeting place and deciding on participants. In the *Overview* phase different roles are assigned to the participant and materials to review are distributed. During the *Preparation* phase the participants review the material, prepare questions and identify possible defects. In the *Meeting*, the actual defects are found, followed by the *Rework* phase, in which the original author of the code resolves the defects. Finally, in the *Follow-up* phase, the moderator of the inspection verifies that all defects have been resolved by the author. If this is not the case, the *Rework* is revisited.

Modern Code Review. MCR (lightweight code reviews) constitutes a more modern approach to CR and represents a variant of the traditional code inspection process. It is characterized as a lightweight process that follows less strict rules and allows for faster and more efficient reviews. Nowadays, MCR is a widely applied practice in both open source and industrial systems [6]. Forms of MCR involve *over-the-shoulder* reviews, *pair programming* and *email-pass-around interviews*. Moreover, MCR is generally supported by additional tools helping developers in reviewing changes by others. These approaches have less overhead compared to formal code inspections and can be equally effective if properly done. Furthermore, MCR is part of the development process and does not require external gatherings such as the Fagan Inspection process, and also allows for historical audits. Additionally, MCR is considered as more time effective and suitable for development teams spread geographically. An example for a tool which is widely used in open-source to support the management of code review processes is *Gerrit* [2], while CheckStyle [1] and PMD [3] are popular tools used for detecting defects (e.g., vulnerabilities [24]) and design issues (e.g., high coupling between objects) in the code under review.

Gerrit Code Review. *Gerrit* is a tool developed and maintained by Google for the Android project, and mostly used in open-source projects. It emerged from a fork of Rietveld[1], which itself was based on Mondrian, a proprietary application used by Google for their code reviews. When working in a team, code usually lives on a central repository (Figure 1.2), where every developer pushes updates to and pulls changes into their local copy of the code. Furthermore, the central repository is often connected to a CI (Continuous Integration) Build server.

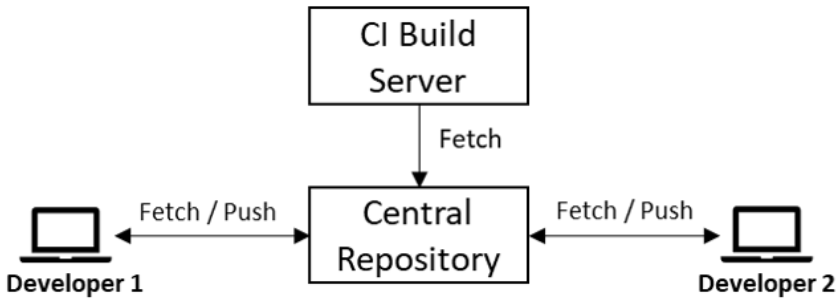

Figure 1.2: General Development Process

Gerrit works with *git*[2] and adds an additional concept, where instead of pushing updates directly to the central repository, updates are pushed to *Gerrit* as *pending changes*, as shown in Figure 1.3. These *pending changes* can only be integrated into the central repository once they have gone through a review and have been accepted[3]. However, the current state of the project is still fetched from the central repository. To review a change, *Gerrit* offers different ways to comment on the uploaded code by an author. In a side-by-side and unified diff view reviewers can leave in-line comments on specific lines or comment on the patch as a whole. For every uploaded patch, reviewers can be assigned and notified. Developers on the team can assume the roles of reviewers and verifiers. Reviewers are tasked with analyzing the code and giving useful feedback in form of comments in order to improve the quality of the submitted patch. Verifiers on the other hand are in charge of making sure that submitted patches serve a clear purpose, are useful and

[1]https://github.com/rietveld-codereview/rietveld
[2]https://git-scm.com/
[3]https://review.openstack.org/Documentation/intro-quick.html

fix defects without breaking the behavior of the existing code base. A new branch is created for every *pending change* under review. This allows for tracking the review and changes made to the patch. Moreover, as every pending change is tracked as a new branch, unit tests can be run before the review. If a patch needs additional changes by the author, a new version of the patch needs to be pushed to review. Only once all changes are approved, the patch is merged into the central repository.

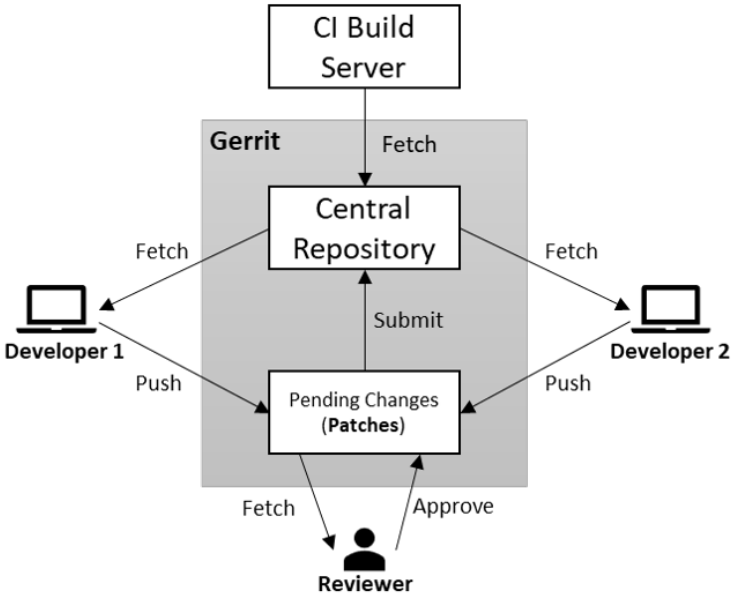

Figure 1.3: *Gerrit* Development Process

1.2 Motivation & Research Questions

The dominant practice in industry is still the manual inspection of the artifact undergoing a change. This is also the case in MCR. As this involves people, CR is often a very effort consuming task of code integration and comes with considerable costs, with developers spending on average six hours per week reviewing changes of others [18]. This is not only substantial in the time spent reviewing, it also forces developers to switch context away from their current work [23]. Although *finding defects* is often stated as the primary expectation of code reviews, this is often not the case. The most common category of comments is regarding code improvements (code practices, unused code and readability). Therefore, functionality issues that should block a code integration are often not found during code reviews [6, 23]. Furthermore, the usefulness of code review comments is largely dependent on people with a certain amount of experience. There is a positive correlation between the reviewer's experience and the overall usefulness of the code review. If the reviewer has no prior experience with the code base, on average only 33% of comments are deemed useful by the author of the change. This ratio increases with every time the reviewer is exposed to the same code base [23]. Another factor that influences the code review workflow is the social aspect. Czerwonka [23] highlights that *"people's roles on the team and their standing in team's hierarchy influence the outcome. Often it is not only the author of the change but*

also the reviewers who find themselves under scrutiny.". Furthermore, the review of the changes can happen as **pre-commit** or **post-commit** reviews (Figure 1.4), each having their own advantages and influences on the outcome of a review. The former requires that changes are reviewed before they are submitted and integrated into the central repository. In the latter, code review takes place only after the changes have been submitted and integrated into the central repository. Pre-commit reviews have the advantage that it ensures the code quality standards set by the team before any work is submitted to the central repository. Furthermore, as code can only be submitted once it has been reviewed, pre-commit reviews ensure that every change is actually reviewed and not postponed or omitted. Finally, the central repository is not affected by eventual bugs that can be found during the code review. A disadvantage of pre-commit reviews is the lowered developer productivity, as each change needs to be reviewed first by other developers, which slows down the development process. In contrast, post-commit reviews do not slow down the development process, as developers can commit changes to the central repository continuously. Moreover, other team members can instantly see changes by other developers and act accordingly. Lastly, post-commit reviews help to understand and verify complex changes, as they might be part of multiple steps and are best reviewed as a whole. Nonetheless, in post-commit reviews, generally more bugs are introduced into the code. Furthermore, as code is not reviewed before its integrated into the central repository, it increases changes for poor code which affects the entire team's work.

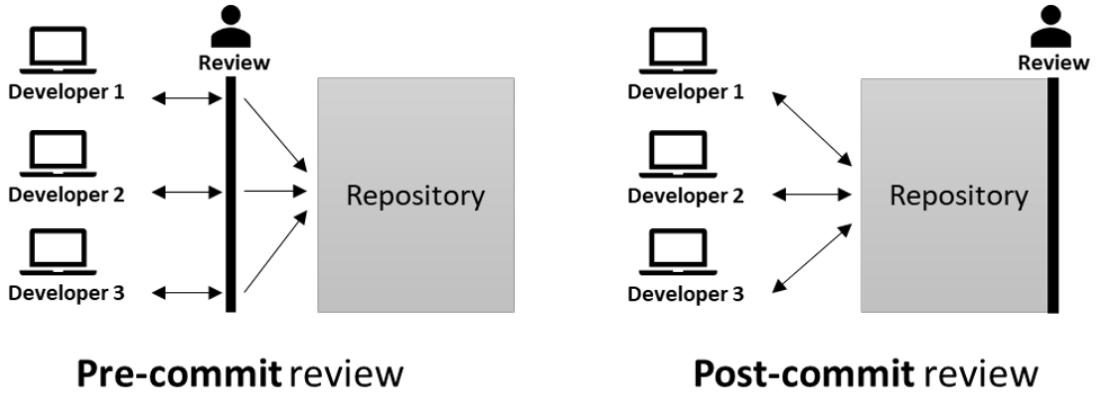

Figure 1.4: Pre-commit and post-commit code reviews.

As stated before, the code review process is often supported by additional tools, such as static analysis tools that provide useful warnings during review or development [55]. Recent research produced further tools to support decisions and actions of MCR in different ways: (i) recommender systems selecting appropriate peer reviewers to evaluate a given patch [8, 52, 73] and (ii) approaches that automatically decompose code review change-sets [9], recommend files to focus on during a review [11] or simply detect potential mistakes [74], e.g., finding inconsistencies between the documentation and the code using natural language processing [75].

A study compared the removal rate of static analysis warnings of PMD[4] and Checkstyle[5] in six different open source projects and found that certain categories of warnings are resolved more often than others in the context of code review. Specifically, warnings concerning *imports*, *regular expressions* and *type resolutions* were removed the most, with an observed removal rate of more

[4]https://github.com/pmd
[5]https://github.com/checkstyle/checkstyle

than 50% up to a 100%. This implies that requiring certain warnings to be resolved prior to code submission to review could reduce the code review effort. Other studies also indicate that certain types of issues are addressed more often than others during code review [6], including missing or incomplete Javadocs and absent license statements [32].

Thesis design and research questions The objective of this thesis is to investigate the approaches and tools that, from a *developer point of view*, are still needed to facilitate MCR activities. Little research investigated (i) the most recurrent or critical code review *change types* developers have to deal with and at the same time (ii) the approaches and tools still needed to auto§mate or accommodate such changes. While previous studies mainly investigated the usage and the limits of existing tools for code review [6, 15, 55, 64], this thesis puts its attention on the specific changes that developers actually perform in code reviews, investigating the possible automation that is needed for supporting such changes. To that end, the following research questions are addressed in this thesis:

- ***RQ_1: What types of changes occur during code reviews?***
  In a first step, a taxonomy of the most critical and recurrent CR *change types* that characterize code reviews was elicited. The taxonomy was designed by performing three steps: (i) an initial version of the taxonomy was generated by qualitatively and quantitatively analyzing 362 review changes and 631 review comments of 10 open source projects; then (ii) topics and CR *change types* emerging from existing taxonomies available from the literature were integrated into this initial taxonomy; finally, (iii) 52 developers were surveyed to integrate eventual missing code review *change types* into the taxonomy.
- ***RQ_2: What are the emerging automation needs of developers in MCR?***
  In a second step, the data, approaches and tools that developers would need to accommodate the identified CR *change types* are investigated. To this end, survey participants were asked to specify (i) the most critical and important review *change types* they usually perform in MCR; and (ii) the type of automation they would need or envision to accommodate these review *change types*. Survey results are then analyzed and promising solutions toward the automation of MCR are identified.
- ***RQ_3: What approaches are feasible for the automation of MCR?***
  By analyzing the feedback given by the participants in the survey, approaches towards an automation of MCR are investigated in a practical way. More specifically, many developers in the study suggested the usage of machine learning in combination with metrics to model a patch submitted to code review. This would allow, with enough training, to predict eventual future changes in a given patch. To that end, a *proof of concept* is presented showing how relevant metrics, describing a patch in code review, can be extracted and could be used in the future for predicting changes.

This thesis makes the following **contributions**:

- It provides a detailed taxonomy of changes occurring in MCR with respect to issues that could potentially be detected and fixed by dedicated recommender systems.
- It provides recommendations regarding approaches, solutions and techniques to detect and fix such issues in MCR.
- It provides a *proof of concept* considering specific code and change metrics mentioned by the participants of the study that can be used to describe a generic patch submitted to code review. Furthermore, a possible setup of a tool extracting these metrics is proposed for future work with the intention of using these metrics in a machine learning approach to predict specific code changes in code reviews.

- It suggests relevant directions for future work in the area of MCR.

The study participants provided insights on the types of approaches and tools they would need in the context of MCR, sharing recommendations, specifying techniques to employ, and highlighting the data to analyze for building recommenders able to automate code reviews activities. Therefore, this thesis provides more information on the types of code review change developers perform in modern code review, and sheds some more light on the approaches and tools that, from a *developer point of view*, are still needed to facilitate MCR activities. This research investigates the gap between current needs and expectations of practitioners towards MCR and provides suggestions for novel tools and approaches to reduce this shortcoming. The authors of the thesis believe that this is an important step for conceiving tools meeting the increasing expectations of developers, and thus, supporting the vision of fully or partial automation in MCR [6,64].

Structure of the thesis. The thesis is divided into four parts. Chapter 2 presents recent and related work in this research area. Chapter 3 describes the methodology of the thesis, namely the manual analysis of review changes (3.1), the conducted survey (3.2) and the applied approach for a first *proof of concept* (3.3) in regard to the automation of certain aspects of MCR. In Chapter 4 the results of this study are presented and discussed by introducing (i) a detailed taxonomy of critical review changes; (ii) a qualitative evaluation of developers' automation needs in the context of MCR; and (iii) a *proof of concept* based on recommendations gathered from participants in the study. Chapter 5 highlights threats to validity, states recommendations for future work and concludes the paper.

Chapter 2

Related Work

2.1 Modern Code Review Process and Practices

MCR process. To the best of the authors' knowledge, Rigby *et al.* [60–62] are the first that empirically investigated the use of code reviews in open source projects, identifying and stating guidelines that developers should follow (e.g., having small and independent patches, do frequent reviews) in order to make the submitted patches accepted by reviewers. In this context, Weißgerber *et al.* [71] found that, in general, the probability of a patch to be accepted is about 40% and that patches containing few changes usually have a higher possibility to be accepted, while Baysal *et al.* [13] discovered that patches submitted by casual contributors have a higher probability to not be reviewed compared to the patches submitted by core contributors of a project. Nurolahzade *et al.* [51] confirmed these findings and additionally showed the importance for reviewers to identify and eliminate immature patches to alleviate huge backlogs.

Benefits and shortcomings of MCR. Other work focused on how developers perform code reviews in industrial and FLOSS (Free, Libre and Open Source Software) projects [6, 44]. Mantyla *et al.* [44] analyzed the code review activities of industrial and FLOSS projects, discovering that the type of defects fixed in code reviews are in most cases related to non-functional aspects of the software. Bacchelli and Bird [6] studied the code review process across different teams at Microsoft by classifying review comments and surveying developers. They found that code review is not only about finding defects, but also serves as a tool to transfer knowledge, increase team awareness and helps finding alternative solutions to problems. Furthermore, they concluded that the available tools for code review do not always meet developers' expectations. The work of this thesis is very close to the one of Bacchelli *et al.* [6], as this thesis aims to fill the gap between expectations and outcomes of code review tools, (i) by studying the types of changes addressed during code review; and (ii) investigating the automated support that developers need or expect during code reviews activities.

Characteristics of good code review. Recent work studied the relevant social dynamics characterizing the code review process [12, 19, 43, 46]. First of all, McIntosh *et al.* [46] studied developer participation during code review and discovered that the degree of freedom that reviewers have impacts both reviewing environments and software quality, confirming the assumption that badly reviewed patches lead to more post-review defects. Following this line of research, Kononenko *et al.* [43] confirmed the importance of code review participation, exploring the relationships between the reviewers' code inspections and related factors in the large open-source project Mozilla. Their results show that 54% of the reviewed changes introduced bugs in the code, and that reviewer workload, experience and participation impact the quality of the code review

process. Other work identified important aspects impacting software quality during code review activities, separating them in technical and non-technical factors [14, 39]. The work of this thesis represents a continuation of this recent research as we are interested to improve developers' productivity during code review practices, by overcoming the problems related to low developers' participation, with recommender systems automating some of the reviewing activities. Furthermore, researchers investigated the usefulness of reviewer activities by analyzing semantical and textual features of review comments. One study built recommendations and guidelines for future research in the area of natural language processing related to code review [28], whereas in another study textual characteristics of useful review comments are extracted and used in a prediction based approach to determine if a comment can be considered useful [58]. Other research looked at the notion of fair reviews, stating that code reviews are not always prioritized consistently and create bias, which can be considered negatively by developers. [20, 33, 42]

Fixed issues in MCR. Similar to the work conducted in this thesis is the work of Mantyla [44] and Beller *et al.* [15], which investigated the defects and problems developers actually fix during code reviews. Following the work of Mantyla, which classified defects found in nine industrial (C/C++) and 23 student (Java) projects into a defect classification, Beller *et al.* [15] manually classified over 1,400 changes taking place in reviewed code from two OSS projects into a validated categorization scheme, classifying them into *evolvability changes* and *functional changes*. According to their findings, 7-35% of review comments are discarded and 10-22% of the changes are not triggered by an explicit review comment. Their results also support the findings by Mantyla [44], that 75% of changes are maintainability issues and only 25% of them are related to functional changes.

Whereas these works investigated the benefits, characteristics and issues of MCR, none have considered at the same time (i) the most recurrent and critical code review *change types* developers deal with in MCR, (ii) the expectations developers have towards code reviews and (iii) the approaches and tools still needed to automate and accommodate such changes. Furthermore, while other works provided insights how MCR can be supported by tools, *e.g.*, static analysis tools [55], this thesis takes a *developers' point of view* and highlights how the gap between the expectations and the outcome of code reviews can be lessened by providing techniques to apply, data to analyze and approaches to follow with the goal of partly or fully automated code reviews.

2.2 Automation in Modern Code Review

Recent research proposed tools and approaches to automate or facilitate some decisions and actions during code reviews [8, 9, 21, 36, 52, 55, 65, 67, 73, 74], as well as methods on how to evaluate these tools and strategies [37].

Static analysis tools in MCR. The use of static analysis (SA) tools to find defects, whether or not they may cause failures, is a common practice for software developers [30, 41, 66, 70] and recent research investigated its usage in the context of code review [55,67], whereas other research investigated the use of SA tools in software engineering in general [16, 72]. Kim and Ernst [41] studied the removal rate of warnings produced by Automated Static Analysis Tools (ASATs) in 3 different projects. Their initial finding suggests that less than 10% of warnings are removed by bug fix changes. In the following, they investigated a prioritization algorithm on historic bug-fix data in code repositories to determine which warnings can be considered important in the context of bug fixing, showing that many of the warnings produced by ASATs are false-positives. Panichella *et al.* [55] investigated the removal rate of ASATs warnings in six *Gerrit* repositories and found in their study that certain warning types produced by these tools are removed more often

than others code review, reporting removal percentages between 50% up to a 100%. Examples of these warnings are *imports*, *regular expressions* and *type resolutions*, thus showing that the review effort can be reduced if the removal of certain warnings is enforced. Beller *et al.* [16] demonstrated that ASATs are popular tools used in various projects but usually do not follow strict policies on their usage. Furthermore, the study showed that the configuration of these tools hardly ever changes after the tool's introduction into the development process. Following this line of research, Zampetti *et al.* [72] studied the usage of ASATs in 20 Java open source projects in the context of continuous integration. Their results showed that a small percentage of broken builds are caused by problems caught by ASATs and that missing adherence to *coding standards* is the main cause behind these broken builds. Furthermore, Vassallo *et al.* [67] demonstrated that SA warnings have different importance to developers depending on the context they are used in. These contexts involve the local environment, code review and continuous integration. Their results show that in the local environment, warnings regarding *code structure* and *logic* are deemed important, whereas in code review warnings related to *style conventions* and *redundancies* are the most focused on. Finally, during continuous integration, warnings about *error handling*, *code logic* and *concurrency* are the main focus.

Advanced strategies to MCR. Other approaches haven been proposed by researchers to support coding or collaborative activities concerning the code review process [8, 9, 11, 47, 53, 73].

Firstly, multiple studies provided approaches to automatically recommend a suitable reviewer for one's code change [36, 52, 65, 73]. Thongtanunam *et al.* [65] investigated the assignment of reviewers in distributed software development and its impact on the review time. They implemented a file location-based code-reviewer recommendation approach based on the similarity of previously reviewed file paths, showing that the overall code review process can be sped up if appropriate reviewers are assigned to a review. Similarly, other studies proposed reviewer recommendations in FLOSS projects based on modification and review expertise [36], based on expertise and collaboration in past reviews using Genetic Algorithm [52] or based on the change history of source code lines [8]. Zanjani *et al.* improved upon this work by leveraging specific information in previously completed reviews instead of using generic review information such as similar source code files or path names.

In addition, to help both reviewers and authors in the development of code and reviewing activities, Barnett *et al.* proposed an approach to automatically decomposes code review changesets [9] into smaller partitions for easier reviews. Baum *et al.* investigated how a code review tool should order and prioritize the different parts of a code change to provide the most support to the developer. They proposed a strategy to recommend the files to focus on during a review by grouping and ordering related changes [11]. Finally, Zang *et al.* [74] presented an interactive approach that can summarize similar code changes and detects potential mistakes by matching a generalized template against the code base. Their results show that by using this approach reviews can be completed more time efficient.

Other approaches and tools developed and suggested by researchers include (i) an approach that leverages semantic differential analysis between code versions to additionally provide developers with behavioral diffs, facilitating the understanding of the underlying reasons of the changes [47]; (ii) a tool that generates code review comments, offering design suggestions based on historic changes in source code repositories [21]; and (iii) tools that provide detailed code review changes (snapshots) of thousands of patches for further processing [53].

Chapter 3

Methodology

The following chapter describes the methodology applied in the thesis. In addition, Figure 3.1 shows the different steps of the research approach. Section 3.1 explains how data for this research was obtained, preprocessed and classified into taxonomies and code review *change types*. Section 3.2 presents the design and structure of the conducted survey with 52 participants, whereas Section 3.3 presents the practical approach taken to a *proof of concept* towards the automation of MCR.

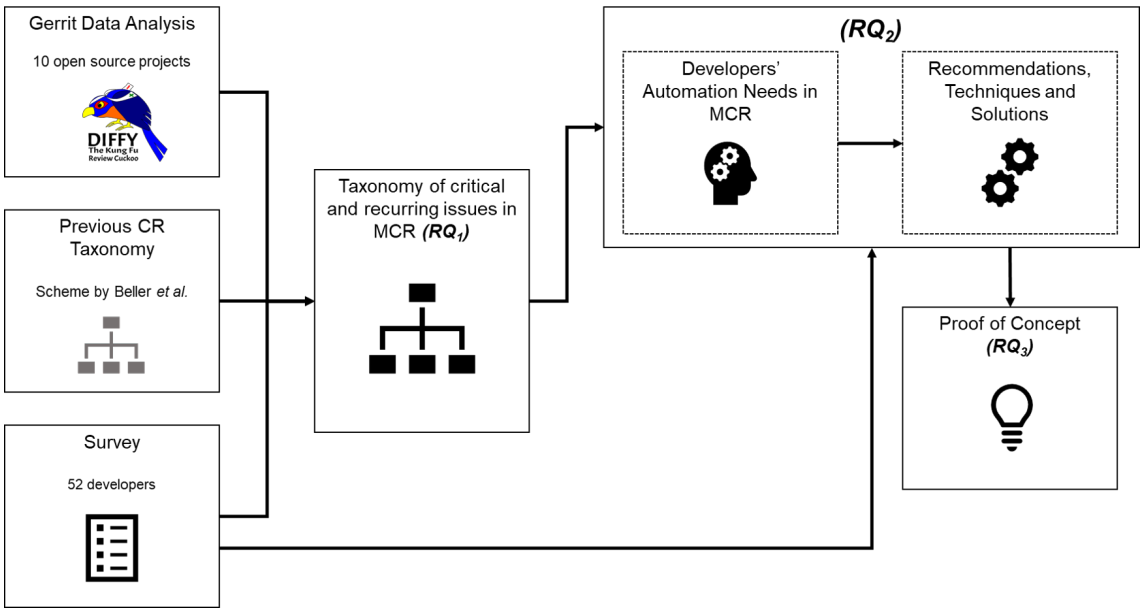

Figure 3.1: Overview Research Approach

3.1 Manual Classification and Validation

Goal and project data. A first step toward answering what types of changes occur in code reviews (RQ_1) was the elicitation of an *initial taxonomy* by analyzing changes in open source projects.

To gain a better understanding about the specific code review *change types* occurring in MCR, the code review comments and historic changes related to ten Java open-source projects were collected, namely data from the following projects:

- **Eclipse Acceleo**[1]: An open-source code-generator from the Eclipse Foundations which allows people to build applications in a model-driven fashion.
- **Eclipse CDT**[2]: C/C++ Development Tools that add support for C/C++ syntax highlighting, debugging, project structures and code formatting.
- **Eclipse Amalgam**[3]: An Eclipse top-level project focused on and aimed at the evolution of model-based development technologies. Its main goal is to rework the user experience related to modeling.
- **Eclipse BPEL**[4]: A project designed to add support to Eclipse for the definition, authoring, editing, deploying, testing and debugging of WS-BPEL (Web Services Business Process Execution Language) processes.
- **Eclipse Cbi**[5]: A Maven project with the Tycho[6] plugins. The Tycho plugins tell Maven how to build Eclipse plugins and OSGi[7] bundles. Many other projects can be considered part of Eclipse Cbi such as Gerrit [2] and Bugzilla[8].
- **Eclipse EGit**[9]: Enables the usage of the Git version control system[10] in Eclipse. The Eclipse EGit project develops tooling for Eclipse on top of JGit[11], which is a Java implementation of Git.
- **Eclipse PDE**[12]: The Plug-in Development Environment (PDE) provides tools to create, develop, test, debug, build and deploy Eclipse plug-ins, fragments, features, update sites and RCP products.
- **Eclipse Egit-Training**[13]: Training platform for Eclipse contributors.
- **Eclipse Jgit:** A lightweight Java library implementing the Git version control system.
- **Eclipse M2E**[14]: A project that provides integration for Apache Maven into the Eclipse IDE.

The observed time period, number of reviews, number of review comments and number of KLOC (Kilo of Lines Of Code) analyzed are reported in Table 3.1. These projects were mainly chosen due to the availability of review information in Gerrit (e.g., evolution of patches, stored CR commits, reviewer comments in patches, etc.) and their different domain and size.

[1] https://www.eclipse.org/acceleo
[2] http://www.eclipse.org/cdt/
[3] http://www.eclipse.org/modeling/amalgam/
[4] http://www.eclipse.org/bpel/
[5] https://git.eclipse.org/r/cbi/org.eclipse.cbi
[6] https://www.eclipse.org/tycho/
[7] https://www.osgi.org/
[8] https://www.bugzilla.org/
[9] http://www.eclipse.org/egit/
[10] https://git-scm.com/
[11] https://www.eclipse.org/jgit/
[12] http://www.eclipse.org/pde/
[13] https://git.eclipse.org/r/sandbox/egit-training
[14] https://git.eclipse.org/r/m2e/m2e-core

Table 3.1: Characteristics of the analyzed projects.

Project	Observed Period	# of review changes	# of reviews comments	# of KLOC
Acceleo	2015-03–2017-03	56	243	622
Amalgam	2015-07–2017-03	4	4	26
Bpel	2012-10–2012-12	1	2	219
Cbi	2015-07–2017-03	1	1	13
Cdt	2012-05–2017-03	70	192	1,600
Egit-github	2012-02–2017-07	25	55	200
Egit-pde	2012-02–2012-03	1	17	531
Egit-training	2012-03–2016-03	3	3	195
JGit	2012-09–2017-03	1	1	212
M2e	2014-03–2017-03	49	130	3
Total	-	**211**	**648**	**3621**

Gerrit Web Application. Each *pending change* in the Gerrit review system is part of a *ChangeId*, which holds together all revisions belonging to a change. The first push to Gerrit creates such a *ChangeId*. Every subsequent push of changes with the same *ChangeId* creates a new revision and branch in the same *ChangeId*. Only once all changes are approved, the patch is merged into the central repository. The revision number of the last reviewed revision is used as the commit-id in the central repository. During the review process, assigned reviewers and verifiers can place comments inside the files or on the whole patch. Reviews can be inspected in the Gerrit Web Application (GWA), where all information regarding a change is summarized.

As shown in Figure 3.2, the following information is available: (1) author of the change, (2) commit-number of the current review, (3) Change-Id, (4) information about the assigned reviewers and verifiers, (5) changed files and (6) information whether comments were placed inside the files by the reviewer.

The GWA also provides side-by-side diff viewing as shown in Figure 3.3 to examine code changes introduced with the patch: (1) full path of the current file under review, (2) patch number to diff against, (3) in-line review comment and (4) buttons to switch between files.

Data extraction. During the review process *Gerrit* stores information about the whole review such as the change-id, files belonging to this review (patch sets), file modifications, review scores given during the review and reviewer comments (in-line comments or general comments on the patches). To obtain all relevant information needed for the research goals of this thesis, *Gerrit* review data was downloaded using the *gerrit query tool*[15]. The tool allows to access the database of *Gerrit* review data with ssh queries, returning a log file in either json or text format containing the specified values. Listing 3.1 represents an example of such a query for the M2E project and Figure 3.4 shows an extract of the obtained log file.

[15]https://git.eclipse.org/r/Documentation/cmd-query.html

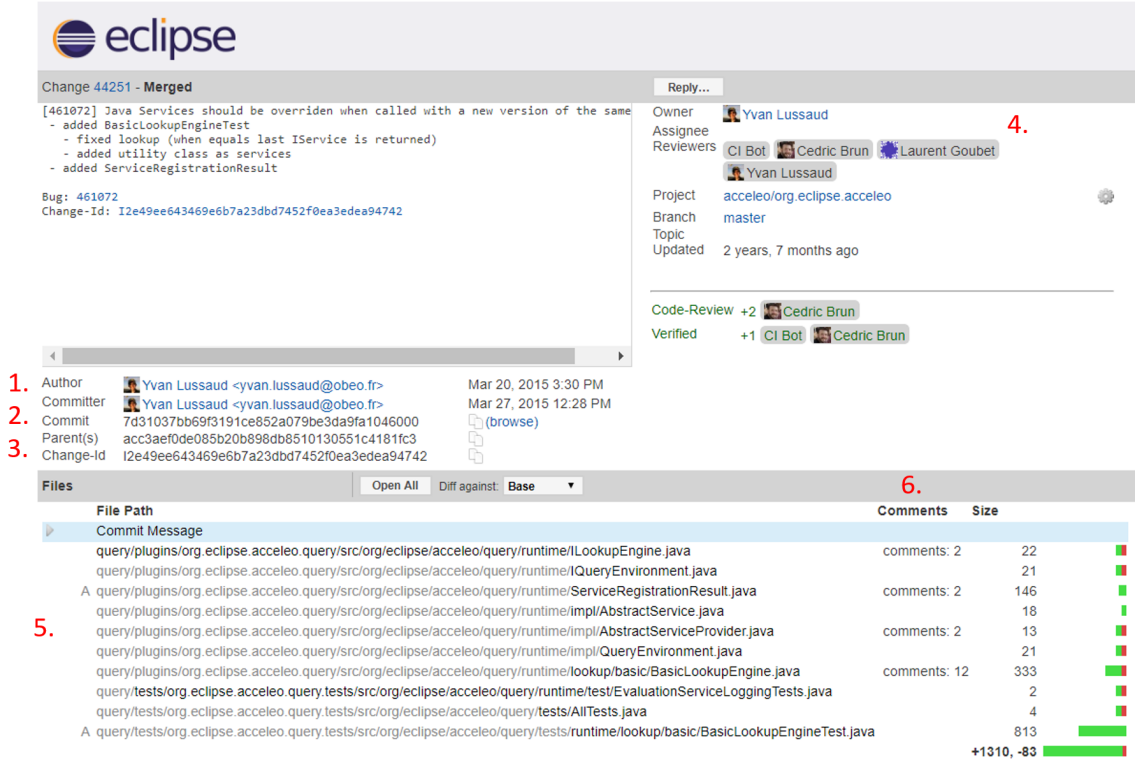

Figure 3.2: Example Gerrit (https://git.eclipse.org/r/#/c/44251/)

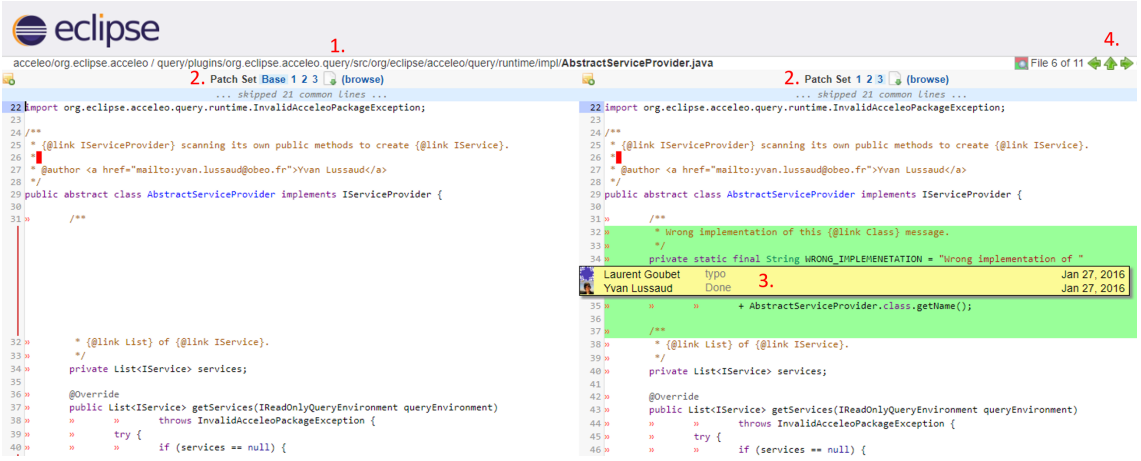

Figure 3.3: Example Gerrit side-by-side diff view (https://git.eclipse.org/r/#/c/44251/)

```
ssh -p 29418 nzaugg@git.eclipse.org gerrit query --format text
project:m2e/m2e-core --patch-sets --current-patch-set --all-approvals
--files --comments --commit-message --dependencies --submit-records
> Logm2e-m2e-core.csv
```

Listing 3.1: Example Gerrit Query

```
change Ieb231142368927ed9d138afa400f1ad1cec175a1
  project: m2e/m2e-core
  branch: master
  id: Ieb231142368927ed9d138afa400f1ad1cec175a1
  number: 128622
  subject: Bug 538461 - MavenBuilder scheduling rule could be null
  owner:
    name: Mickael Istria
    email: mistria@redhat.com
    username: mistria
  url: https://git.eclipse.org/r/128622
  commitMessage: Bug 538461 - MavenBuilder scheduling rule could be null

                 Added an experimental and hidden (not visibile in UI) preference to make
                 the Maven builders schedulingRule null.

                 Change-Id: Ieb231142368927ed9d138afa400f1ad1cec175a1
                 Signed-off-by: Mickael Istria <mistria@redhat.com>
  createdOn: 2018-09-04 04:53:01 EDT
  lastUpdated: 2018-09-05 04:21:18 EDT
  open: true
  status: NEW
  ...
  comments:
    timestamp: 2018-09-04 15:39:41 EDT
    reviewer:
      name: Fred Bricon
      email: fbricon@gmail.com
      username: fbricon
    message: Patch Set 1: Code-Review-1
             (4 comments)
             Please add checkbox in the preference page to expose that new feature (parallele builds, don't mention null rule)
             The checkbox would only show up in the UI if eclipse is started with -Dm2e.showExperimentalFeatures=true.
             That way it'll be easier to play with (by adding the flag to eclipse.ini), while still being hidden to the masses
```

Figure 3.4: Example Gerrit Log for the M2E Project

From the obtained log file, the following files in Comma Separated Values (CSV) format were created for further processing:

- *<project>-LogGerritCommentsInsideReview.csv*: contains in-line comments in files of the available patchsets as well as line numbers and corresponding file paths.
- *<project>-LogGerritPatchSets.csv*: contains all relevant information regarding the patch sets such as the change-id the patches belong to and approval scores.
- *<project>-LogGerritComments.csv*: contains general comments placed on whole patch sets during the review.

Data classification into an initial taxonomy. The data files were then preprocessed by considering only patches that were merged into the central repository as well as by pruning out patches that only contained auto-generated comments such as "Done". Auto-generated reviewer comments are added by accepting an in-line comment inside the GWA. The authors then selected among all code review commits or changes in the dataset the ones reporting explicit reviewers' comments on the quality of patches. This step was needed to investigate actual changes triggered by comments of reviewers. Hence, two authors of this work manually analyzed **648** reviewers' comments related to **211** CR commits, by applying grounded theory [69], focusing on the comments that could be relevant to the study (i.e., comments mentioning desired changes

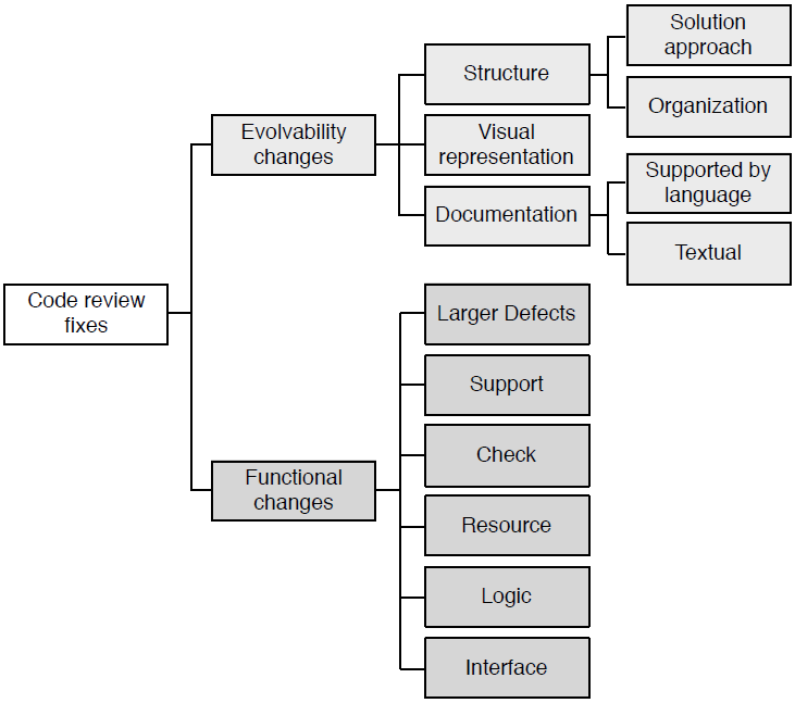

Figure 3.5: Taxonomy of changes of code under review by Beller *et al.* [15]

or highlighting issues in the code). By doing so, the two authors also verified whether the CR changes performed by in the patches actually addressed the reviewers' comments. For each analyzed change, a short description and potential CR *change type* was noted. If the description did not match a previously reported *change type*, a new CR *change type* was added; else it was merged with an existing *change type*. Then, the reported *change types* were used to build the *initial taxonomy* containing 3 high-level and 15 low-level categories of *change types* in MCR. This *initial taxonomy* laid the basis for the future investigations in this thesis. The detailed *initial taxonomy* will be introduced in Chapter 4.

Validation with an existing taxonomy. In order to validate the *initial taxonomy* and find eventual missing categories, an existing taxonomy from the literature was consulted. As mentioned in the chapter Related Work, Beller *et al.* [15] manually analyzed changes taking place in reviewed code from two OSS projects and classified them into *evolvability changes* and *functional changes*, as shown in their validated categorization scheme (Figure 3.5). To verify the completeness of the initial taxonomy, emerged via manual analysis of core review data of the projects reported in Table 3.1, a one-to-one matching was performed between elements in the *initial taxonomy* and the elements of the categorization scheme by Beller *et al.* [15]. The authors observed that some CR *change types* composing the *initial taxonomy* were also present in the one by Beller *et al.*, while others were not included. Thus, categories coming from the scheme by Beller *et al.* and the previously elicited categories were split, merged, refactored, integrated and combined into an *intermediate taxonomy*. Further details about the *intermediate taxonomy* are presented and discussed in Chapter 4.

Validation through a survey. In a further step to build the *final taxonomy* of detailed CR *change types*, the *intermediate taxonomy* was validated through a survey (Section 3.2), asking developers

to state whether they consider the *intermediate taxonomy* to be exhaustive (RQ_1). Furthermore, the survey asked participants what type of feedback they usually receive or expect in code reviews (RQ_1). By evaluating the answers given in the survey, the *intermediate taxonomy* was extended with additional *change types*. The output of this phase consisted of a *final taxonomy* that was named CRAM (**C**ode **R**eview ch**A**nges **M**odel).

As shown in the work by Beller *et al.* [15], most changes (75%) in MCR are maintainability-related problems and only 25% are considered functional issues. Following this line, to quantitatively discuss recurring changes in MCR, one author of the thesis **re-classified** all reviewer comments in the preprocessed dataset into the *final taxonomy* CRAM. A second author validated this classification by selecting 100 random samples from the dataset, classifying these samples and ultimately comparing them with the classification by the other author. Whenever an author found a classification which differed from the other author's classification, they inspected the detailed reviewer comment in the GWA and discussed it. In this way, eventual misclassification possibilities were minimized. The classified dataset was then quantitatively analyzed to learn more about recurrent *change types* in MCR.

3.2 Survey

Goal. As already mentioned, the first part of the survey aimed at validating the *intermediate taxonomy* and finding out what type of feedback developers receive and expect in MCR. Furthermore, the survey asked participants to state (i) what tools they need or envision to support relevant CR changes (RQ_2); and (ii) which approaches developers suggest to solve the recurrent issues in CRAM (RQ_3).

Design. The survey was implemented using *Google Forms*[16]. The structure of the questionnaire consisted of 18 questions, which included six multiple choice (MC), and twelve open (O) questions. In order to receive less biased answers from the participants, many were posed as open questions, thus giving the developers opportunity to give more detailed and personalized feedback.

Table 3.2: Survey questions. (MC: Multiple Choice, O: Open answer)

Section	ID	Summarized Question	Type	# Resp.
Background	Q0.1	What is your current job?	MC	52
	Q0.2	Approximatively, what is the size (in terms of lines of code) of the system you are contributing in most?	O	52
	Q0.3	What is the approximate size of the development team of the system you are contributing to most?	O	52
	Q0.4	How many years of programming experience do you have?	MC	52
	Q0.5	How do you rate your programming experience?	MC	52
Taxonomy Evaluation	Q1.1	What is a code review?	O	52
	Q1.2	Does the taxonomy cover all changes that occur in code reviews?	MC+O	52
	Q1.3	Which Change categories/Topics occur the most inside code reviews?	O	52
	Q1.4	What kind of feedback do you expect from other developers during code reviews?	O	52
	Q1.5	What kind of feedback do you usually receive from other developers during code reviews?	O	52
Automation Needs	Q2.1	What kind of feedback would you expect from recommender-tools during code review?	O	52
	Q2.2	What kind of automation do you envision for automating code review practices?	O	52
	Q2.3	What kind of automation do you envision for the fixing and detection of Documentation issues?	O	52
	Q2.4	What kind of automation do you envision for the fixing and detection of Style issues?	O	52
	Q2.5	What kind of automation do you envision for the fixing and detection of Structural issues?	O	52
	Q2.6	Which code review change types could be automatically detected and/or fixed by tools?	O	52
	Q2.7	How would you approach the detection and fixing of the code review change types mentioned in Q2.6?	O	52

The survey questions reported in Table 3.2 were grouped into three topics: (i) *Background*, (ii) *Taxonomy Evaluation*, and (iii) *Automation Needs*. Questions in the *Background* section asked partic-

[16]https://gsuite.google.com/products/forms/

Table 3.3: Information About Survey Participants

Participants Profile	Nr. (%)
Industrial Developer	**50%**
Open Source Developer	11.15%
Senior Researcher	19.2%
CS Student	9.6%
Other	9.6%

Team Size		Projects Size [LoC]	
1-5	**38%**	1,000-300,000	**66%**
5-10	14%	300,000-1,000,000	15%
10-15	10%	>1,000,000	19%
>15	**38%**		

Experience (Years)		Experience (Rate)	
< 2	1.9%	Poor	1.9%
2-5	11.5%	Fair	0%
5-8	19.2%	Good	19.2%
>8	**67.3%**	Very Good	32.7%
		Excellent	**46.2%**

ipants about their occupation as well as their experience with programming and CR in general. The questions in the other two sections, *Taxonomy Evaluation*, and *Automation Needs*, represented the core part of the survey, aiming at understanding code review practices and related automation needs and approaches.

Background. The survey was available for two months to maximize the amount of collected answers and more than 200 direct contacts were invited to fill out the questionnaire. In total 52 responses were collected, with a return rate of about 23%. Table 3.3 lists demographic information about the participants in the survey. Among the participants, there were 26 (50%) industrial and 6 (11.5%) open-source developers (Q0.1). The rest of the participants (38.5%) consisted of Senior Researchers, Computer Science students and other occupations. In regard to the self-estimation of their own development experience, most of the developers rated themselves as “very good” (32.7%) or “excellent” (46.2%) programmers (Q0.5), and 21% rated themselves as "good", "fair" or "poor" programmers. Moreover, most of the participants are experienced developers, with around 30% of them having between 2 and 8 years of development experience, and around 67% having even more than 8 (Q0.4). Only 1.9% of the participants have less than two years of programming experience.

Taxonomy evaluation. The *Taxonomy Evaluation* questions were aimed at assessing the completeness of the *intermediate taxonomy* (RQ_1) as well as the type of feedback developers usually receive and expect in code reviews (RQ_1). To this extent, contextually to the five questions of section *Taxonomy Evaluation* (Q1.1-Q1.5), the participants were given access to the *intermediate taxonomy* derived after the manual classification of changes and the integration with Beller’s taxonomy [15]. At this stage of the survey developers had the possibility to evaluate the *intermediate taxonomy* and to suggest further categories to integrate into it (Q1.2). Questions Q1.3-Q1.5 asked about the feedback developers usually expect and receive by other reviewers in MCR. For the assessment of the saturation of the taxonomy one of the authors performed an iterative content analysis [40]

of the feedback provided by the participants. Thus, the author started with an empty list of CR *change type* categories and carefully analyzed the feedback provided by the developers. Each time a new CR *change type* category was found, the category was added to a list and the feedback was labeled with the matching categories. The labeled feedback is available in the appendix. After this step, the initial categorization was refined, performing another iteration involving the other author double-checking the classification of the feedback and removing potentially redundant categories from the list. In combination with the *intermediate taxonomy*, the new categories were summarized into CRAM, a taxonomy of CR *change types* grouped in high- and low-level categories.

Automation needs. The *Automation Needs* section (Q2.1-Q2.7) was focused on understanding which tools developers would need during code review (Q2.1 and Q2.2), with particular focus on *recurrent* and *critical* changes and problems occurring in CR activities (Q2.3-Q2.5) [55]. Moreover, questions Q2.6 and Q2.7 aimed at understanding how developers would approach the automatic detection and fixing of CR *change types*.

Automation needs evaluation process. For the evaluation of the *Automation Needs* the authors conducted an iterative content analysis [40] of the feedback provided by the participants. In a first step the answers were downloaded from *Google Forms* and each question in Q2.1 to Q2.7 was separated into its own list. The next step involved one author going through each list and reading the feedback. By doing so, the author highlighted sentences or words which referred to a category of CRAM or provided interesting information for the evaluation. If a feedback contained information about multiple CRAM categories, it was duplicated to reliably track the number of times certain categories were mentioned. Feedback that was cryptic or ambiguous was encoded as such and not considered for the evaluation. In a next step, the duplicate mentions by participants were merged for the final evaluation step, leaving only the correct number of times participants mentioned a CRAM category. The full dataset with the encoded survey feedback is available in the appendix. This procedure was applied for questions Q2.1 - Q2.5 and Q2.6 - Q2.7 individually, the former being more focused on automation needs in MCR and the latter asking questions about enabling such automation needs.

Table 3.4 shows an example of how the feedback for Q2.3 of a participant was evaluated and split into multiple rows. In this way, **364** answers to **7** open questions were analyzed and encoded. To gain more insights into the various needed automation solutions and approaches mentioned by developers, clusters of similar statements were built. This facilitated the evaluation of how frequent certain solutions were mentioned.

Table 3.5 shows an example of how different suggestions were given within the same CRAM category. This is illustrated on the low-level categories *Object-Oriented Changes* and *Comments*. Within these categories, multiple clusters were introduced: *detect architectural violations* and *suggest design patterns* for *Object-Oriented Changes* and *generate Javadoc comments* and *propose comments (templates)* for the *Comments* category.

3.3 Proof of Concept

In order to investigate the suggested strategies by developers on how to implement and automate certain aspects of MCR, a *proof of concept* is elaborated (RQ_3).

Strategy. An approach is proposed how CR activities could be facilitated by tools that are able to detect and fix issues in MCR. To that end, the results of RQ_2 were consulted and analyzed. The approach is based on the solutions, techniques and data developers suggested in the survey.

Table 3.4: Encoding of Survey Feedback

High-level	Low-level	Feedback	Question	Question ID	Occupation
Documentation	Naming	*"in Terms of Java, I would expect that the tool could help generating the javadocs, the license header, **fix the naming conventions** if possible, propose some comments (maybe templates), suggest the right modifiers. Pretty much everything."*	Q2.3	automation_doc_14	Senior Researcher
Documentation	Visibility (Modifiers)	*"in Terms of Java, I would expect that the tool could help generating the javadocs, the license header, fix the naming conventions if possible, propose some comments (maybe templates), **suggest the right modifiers.** Pretty much everything."*	Q2.3	automation_doc_14	Senior Researcher

Table 3.5: Clustering of Survey Feedback

High-level	Low-level	Solution	# distinct Mentions
Structure	Object-Oriented Changes	detect architectural violations	2
Structure	Object-Oriented Changes	suggest design patterns	1
....			
Documentation	Comments	generate Javadoc comments	1
Documentation	Comments	propose pomments (templates)	1
...			

A *proof of concept* was designed to explore how various metrics that characterize a patch can be extracted in the MCR process. The *proof of concept* highlights how an implementation towards the extraction of these metrics can look like.

Feature and metrics selection. Based on developer feedback in the survey (Section 3.2) regarding automation needs in MCR (RQ_2), the proposed heterogeneous types of software artifacts (e.g., commit notes, source code and Javadoc documentation) and historical change data were analyzed. Based on this information, various *CR metrics* were conceived that characterize the status of a patch in any given *CR commit*. More specifically the following types of metrics:

- *Static Code Metrics* such as Lines of Code (LOC) and Cyclomatic Complexity.
- *Change Metrics* describing characteristics of the review process such as the age of a file or how many authors have previously changed the file.
- *Object Oriented Metrics* using the metrics suite by Chidamber-Kemerer [22] such as Depth of Inheritance Tree and Number of Fields.
- Metrics built around the output of ASATs. Considered were warnings by Checkstyle and PMD.

- Tf-idf frequency of word in commit messages of submitted patches to review.
- Low level source code changes obtained by the existing tool CHANGEDISTILLER [31].

The definition of these metrics (i) was guided by the *recommendations* received by the participants and (ii) conceived around the specific CR *change types* in CRAM. Chapter 4 presents more information about the metrics and discussed their characteristics.

Chapter 4

Results and Discussion

In this chapter, results obtained in the manual classification of review changes and the evaluation of the survey are presented and discussed. More specifically, in Section 4.1, CRAM is introduced, which constitutes a detailed taxonomy of code changes occurring in MCR (RQ_1). Section 4.2 presents the evaluation of the feedback gathered in the survey regarding *automation needs* in MCR (RQ_2). In Section 4.3, the metrics considered for a *proof of concept* are defined and presented (RQ_3).

4.1 Change Types in MCR

In the following the results obtained during the initial manual classification of review changes and during the evaluation of survey questions Q1.1-Q1.5 (*Taxonomy Evaluation*) are presented and discussed, answering RQ_1.

4.1.1 Initial Taxonomies

Initial taxonomy. After the initial manual classification of CR comments and commits, a total of ***15 initial potential CR change types*** were defined and grouped into an initial taxonomy of 3 high- and 15 low-level categories of changes reported in Table 4.1. The upper half of the taxonomy represents changes analyzed in source code files in code review and are subdivided into either *production* or *test code* of a project. Each change can then be categorized as either *corrective*, *perfective* or *other*. In this first taxonomy, a corrective change was only classified as being a *bug-fix* or a *change in the documentation* of the code. Perfective changes on the other hand are related to changes which do not change the functionality of the system, but rather improve certain parts of it (*Code Documentation, Style, Refactorings, Best Practice, Performance, License Header, Testing*). The high-level category *Other Changes* reports additional changes that were encountered during the manual classification of the review comments. Reported are changes regarding *Continuous Integration/Continuous Deployment* (CI/CD), changes specific to a programming language or framework (e.g., *pom.xml* for Maven [1]), configurations for *Automated Static Analysis Tools* (ASAT), e.g., Checkstyle [1], changes regarding general *software documentation* and *commit messages*.

Integrated taxonomy. As described in the Methodology (Chapter 3), the already existing taxonomy by Beller *et al.* was consulted to integrate *change types* previously not considered. In doing so, it was noticed that most of the categories in the *initial taxonomy* (Table 4.1) were also present in Beller's. Thus, Beller's categories and the ones defined during the manual classification were integrated, merged and combined into an *intermediate taxonomy* which can be seen in Tables 4.2 and

[1] https://maven.apache.org/

Table 4.1: Initial taxonomy of changes in CR

Artifact	Maintenance Activity	Maintenance Topic
Production/Test Code	Corrective	**Code Documentation:** issues in Javadoc, modifier declarations and spelling mistakes
		Bug-fix: all changes involved in fixing the functionality of the source code (i.e., missing checks, logic etc.)
	Perfective	**Code Documentation:** issues in Javadoc, modifier declarations and spelling mistakes.
		Style: changes regarding the formatting, white space usage, indentation, blank and long lines
		Refactorings: improvements needed to the organization of the software elements (i.e., splitting an interface into two separate interfaces or introduction of design patterns)
		Best Practice: issues related to violations of standard coding conventions
		Performance: issues regarding the optimization of the software (i.e., resource and memory usage)
		License Header: missing license statements in source code files
		Testing: missing tests, lack of test coverage and general issues in test classes and functions
Other Changes		**Continuous Integration / Continuous Deployment configurations:** changes to configuration files concerning CI or CD pipeline/setup
		Language or Framework specific: changes to files native to the used programming language, e.g., MANIFEST for Java
		Automated Static Analysis Tools Configurations: changes in the configuration of Linters, Checkers and Recommenders used in the project (e.g., Checkstyle, PMD, FindBugs etc.)
		Software Documentation: changes to the external Software Documentation files
		Commit Message: updates/changes in the commit message of a submitted patch. Mostly related to wrong description of the change or not capturing all changes
		Other Changes: includes changes go XML, Scripts, README files, HTML files and Version Control

4.3. Where the *initial taxonomy* followed the notion of dividing changes into *corrective* and *perfective* changes, the new taxonomy differentiates between *maintainability* and *functionality* changes. *Code Documentation* in the *initial taxonomy* was split up and integrated into the respective more detailed documentation category by Beller. Additionally, the *License Header* category which was not present in Beller's taxonomy was moved to *Textual Documentation* changes. As *Style* was already present in Bellers taxonomy in more detail, these categories were adopted. Changes falling under *Refactoring* in the *initial taxonomy* were similar to the *Structure* changes by Beller and thus added to the *intermediate taxonomy*. Additionally, the high level category *Functionality* comprising changes to the *Interface*, *Logic*, *Resource*, *Check* and *Larger defects* was added, thus replacing *Bug-fix* in the *initial taxonomy*.

One high level category which was not yet captured by the taxonomy by Beller was the *Other Changes*. In this category *change types* are reported that are usually not found in source-code files, which are nonetheless essential to the runtime of a software project. These changes include changes in *commit messages*, *CI/CD configuration files*, *ASAT configurations*, *language or framework specific changes* and *runtime-configurations*.

4.1.2 CRAM Model

The taxonomy obtained after integrating and merging with the taxonomy by Beller built the basis for the survey and its evaluation. As reported in the survey description (3.2), the study participants were asked to provide feedback on this new taxonomy (Q1.1-Q1.2 in Table 3.2). As a result, 28% of the developers claimed in Q1.2 that the proposed taxonomy was incomplete, reporting a total of 17 answers related to additional activities, tasks or changes occurring during code reviews not yet reported in the provided taxonomy.

Table 4.2: Intermediate taxonomy after validation with the taxonomy by Beller *et al.* [15] - Part 1

Activity	Category	Topic	Detailed Change
Maintainability & Perfective Maintenance	**Documentation (D)**	**Textual Documentation:** Issues concerning the documentation through textual representation, such as naming of classes, method, variables. This also includes license headers, typos in either line comments or Javadoc	**Naming:** problems relating to software element (e.g., methods, classes, variables, etc.) names that do not conform to the naming policy of the project **Comments:** explanations of complex code fragments, classes, methods. Issues include wrongly placed comments, missing comments, missing or wrong Javadoc etc. **License Header:** issues regarding missing or wrong license headers inside source-files **Other:**
		Language Supported Documentation: Documentation through statements/ elements that the programming language offers (e.g., java public modifier to document that it is accessible from the outside)	**Immutability:** not declaring a variable to be immutable when it should have been or declaring it immutable when it should have not been **Visibility (Modifiers):** software element (e.g. method, variable, class) has too much or too restricted visibility
	Style (S)		**Brackets & Braces:** e.g., single statement after a conditional branch **Indentation:** consistent indentation of the code **Blank Lines:** excess of blank lines or too few blank lines or wrong split of lines **Long Lines:** code statement too long, over a specific amount of characters **Whitespace Usage:** usages of blank spaces in the code **Grouping:** grouping of methods with related functionality or adding class variables at the beginning of the class **Commented out code:** remove code that is commented out (also TODO and FIXME)
	Structure (STR)	**Re-implementation:** Structural defects require an alternative implementation method. For example, replacing the program's array data structure with a vector and knowing the existence of prebuilt functionality that could be used instead of a self-programmed implementation would be considered a solution approach defect. Therefore, solution approach defects are not about re-organizing existing code but rethinking the current solution and implementing it in a different way.	**Semantic Duplication:** code structures that have a similar intention but are implemented syntactically different **Semantic Dead Code:** code fragments that are executed, but they do not serve any meaningful purpose and/or have no effect on the result **Change Function:** change function call to another function because it uses old or deprecated functions **Standard Coding Conventions:** use exceptions for error messaging instead of return values, use predefined constants instead of magic numbers, built-in data structures instead of own implementation etc. **New Functionality:** new functionality to ensure evolvability, e.g., create new classes, methods to make code more maintainable **Testing:** issues regarding test coverage, wrong tests, additional tests etc. **Other**
		Organization: Defects that can be fixed by applying structural modifications to the software. Moving a piece of functionality from module A to module B is a possible strategy for this.	**Imports:** issues with wrong or missing or unused import statements **Move Functionality:** move functions, part of functions, or other functional elements to a different class, file, or module **Long Sub Routine:** split long and complex functions into multiple functions **Dead Code:** remove code that is never reached and executed **Duplication / Redundant Code:** remove duplicate code or code that is not used **Complex Code / Simplification:** restructure or rewrite implementation to make it more understandable **Statement Issue:** splitting, combining or otherwise reorganizing a statement inside a function **Consistency:** means the need to keep code consistent in a sense that similar code elements operate in a similar fashion and are more or less symmetrical. For example, similar tasks in similar classes should have similar implementations **Other**
Functionality & Corrective Maintenance	**Interface (I)**		**Function Call:** call to another part of system or library is incorrect or missing **Parameter:** function call or other interaction has incorrect or missing parameters
	Logic (L)		**Compare:** mistake in a comparison statement **Computation:** computations produce incorrect results **Wrong Location:** correct operation is performed, but it is done too soon or too late **Algorithm/Performance:** inefficient algorithm is used
	Resource (R)		**Variable Initialization:** variables are left uninitialized prior to use. Uninitialized variables may contain any value and using such variable for comparison or calculation produces arbitrary results **Memory Management:** mistake is made in handling the system memory **Data & Resource Manipulation:** defects related to manipulating or releasing data or other resources
	Check (C)		**Check Function:** when a function is called there is also a need to check that the value returned is valid and that no error occurred **Check Variable:** there is a need to check variable **Check User Input:** the need to validate user input
	Larger Defects (LD)		**Completeness:** partially implemented feature **GUI:** defects in the user interface code relating to the consistency of the user-interface, and to the options made possible to the user in each situation. **Check outside code:** defects that required that part of the application code that was not under review to be checked, as it was likely to contain incorrect code based on the current review.

Table 4.3: Intermediate taxonomy after validation with the taxonomy by Beller *et al.* [15] - Part 2

Artifact	Activity
Other Changes Changes not typically found in source-code files (.java, .py, .cpp etc.) which are nonetheless essential to the runtime of a project	**Commit Message:** updates/changes in the commit message of a submitted patch. Mostly related to wrong description of the change or not capturing all changes
	Continuous Integration / Continuous Deployment configurations: changes to configuration files concerning the Continuous Integration or Continuous Deployment pipeline/setup
	Automated Static Analysis Tools configurations: changes in the configuration of Linters, Checkers, Recommenders used in the project (e.g., Checkstyle, PMD, FindBugs etc.)
	Language or Framework specific: changes to files native to the used programming language. For example MANIFEST for Java
	External Software Documentation: changes to the external Software Documentation files
	Runtime Configurations: docker-configs, ansible playbooks, deployment configs etc
	Other: includes changes to XML, Scripts, README files, HTML files and Version Control

Additional changes suggested by developers. The encoding of these answers resulted in the identification of a total of ***3 additional change types***, not previously found in the taxonomy (highlighted in BLUE in Tables 4.4 and 4.5). Developers reported that in code review, changes regarding *Logging and Error* handling should be considered in the taxonomy. Furthermore, multiple participants mentioned the importance of the *security aspect* when doing code reviews; something which was not yet considered in the taxonomy. Lastly, the encoding of the feedback revealed that developers think code reviews often result in a change to the system architecture, such as splitting an interface into two distinct interfaces, introducing abstractions, or the inclusion of design patterns. Thus, the change type *Architectural changes* was added to the taxonomy. These categories were then integrated into the final set of CR *change types* composing CRAM. The encoded sentences from Q1.2 can be found in the appendix.

Taxonomy structure. Tables 4.4, 4.5 and 4.6 provide an overview of CRAM. To facilitate the understanding of this taxonomy, each CR change type is grouped according to different high- and low-level dimensions: (i) *artifact type* involved in the change, e.g., test, production code (in Tables 4.4 and 4.5) or configuration files (in Table 4.6); (ii) the performed *type of CR activities/changes* (e.g., perfective and corrective maintenance); (iii) the specific *CR change categories* associated with each activity (e.g., changes related to artifact structure, its logic and resource utilization); and finally, (iv) the detailed or fine-grained *changes* associated with each CR change category. Moreover, the taxonomy highlights with different colors the detailed CR *change types* that emerged during the manual classification and the integration with Beller's taxonomy. Specifically, (i) in **BLACK** CR changes types are highlighted that overlapped or were merged with the schema by Beller *et al.* [15]; (ii) in **RED** categories are highlighted that emerged during the manual analysis of CR commits and comments of the ten open source projects and that were not present in the schema by Beller *et al.*; and (iii) in **BLUE** the additional *change types* are highlighted suggested by the developers and that were not present in the *intermediate taxonomy*.

Taxonomy categories. CRAM includes CR changes related to the *structure*, *documentation* and *style* of the test and production code. Other changes are performed to fix issues related to the way existing or added functionalities are implemented in the patch under review, such as *interface* (issues related to the communication with a different part of the system), the *logic* of the code, its *resource* allocation and consumption, wrong and incomplete *checks* of values assigned to code elements, and different types of *defects*. Table 4.5 reports further CR *change types* related to the modifications made by developers in non-source-code files which are, in some cases, also essential to the runtime of a project: (i) configuration files related to the *continuous integration and continuous deployment* processes, and *static analysis tools*; (ii) *language or framework specific* files;

(iii) *changes to external software documentations*; (iv) files responsible for *runtime configurations* (e.g., Docker files); (v) *commit messages*; and (vi) *other artifacts* (e.g., README files).

CRAM represents the final taxonomy of detailed changes occurring in MCR and represents one of the contributions of this thesis. The findings and insights gained to answer the first research question are discussed in Section 4.1.3.

Recurring issues in MCR. After manually classifying the dataset of review changes into CRAM, the following distribution of changes was obtained. The following tables show number of occurrences of review *change types* in CRAM and their percentages. Table 4.7 shows the distribution of changes on *Artifact*-level. Around 83% of all review comments triggered changes in source code files belonging to either production code or test code. Table 4.8 illustrates the distribution of changes in production and test code and their respective count in the activities *maintainability* and *functionality*. Table 4.9 depicts how overall the changes are distributed within the *functionality* and *maintainability* activities. Changes that do not belong to either production or test code were grouped under *Other Changes* and are presented in Table 4.10. The distribution within the different taxonomy *change types* are shown in Table 4.11. This includes all changes from both Production and Test Code. Table 4.12 depicts the distribution of changes of the top 5 topics and their respective percentage. This includes all changes from both Production and Test Code. Table 4.13 reports numbers on the distribution of the top 15 *Detailed Change* category.

Results gathered from the manual classification regarding the distribution of code *change types* observed in the ten studied projects are discussed together with the findings from CRAM in Section 4.1.3.

4.1.3 Recurrent and Novel Change Types in MCR

In the following, we discuss our results and answer RQ_1: *"What types of changes occur during code reviews?"*.

Knowledge transfer. The evaluation of survey question Q1.1 showed, that most (78%) developers believe contemporary CR practices are needed to facilitate the team knowledge transfer as well as to improve the overall quality and performance of the patch under review. This preliminary finding is in line with the results of the work by Bacchelli and Bird [6]. However, we also discovered that compared to the schema by Beller *et al.* [15], emerging *change types* characterize MCR activities and that novel tools and approaches are needed to support such activities. In the following, the main elements composing CRAM, with specific focus on the emerged CR categories compared to the schema by Beller *et al.* are described and our findings are presented.

> MCR facilitates team knowledge transfer and helps to improve the overall quality and performance of the reviewed patch.

Documentation. Changes/issues related to **Documentation** (D), Style (S) and Structural (STR) are very recurrent in both traditional and MCR as reported by 60% of our study participants. Beller's study [15] suggests that most documentation issues involve changes to (i) missing, wrong or incomplete Javadocs and in-line comments (D.1); and (ii) inconsistent naming choices in the

Table 4.4: CRAM (Code Review chAnges-Model) - Part 1

Activity	Category	Topic	Detailed Change
Maintainability & Perfective Maintenance	**Documentation (D)**	**Textual Documentation:** Issues concerning the documentation through textual representation, such as naming of classes, method, variables. This also includes license headers, typos in either line comments or Javadoc	**(D.1) - Naming:** problems relating to software element (e.g., methods, classes, variables, etc.) names that do not conform to the naming policy of the project **(D.2) - Comments:** explanations of complex code fragments, classes, methods. Issues include wrongly placed comments, missing comments, missing or wrong Javadoc etc. **(D.3) - License Header:** issues regarding missing or wrong license headers inside source-files **(D.4) - Typos:** spelling mistakes in the documentation
		Language Supported Documentation: Documentation through statements/ elements that the programming language offers (e.g., java public modifier to document that it is accessible from the outside)	**(D.5) - Immutability:** not declaring a variable to be immutable when it should have been or declaring it immutable when it should have not been **(D.6) - Visibility (Modifiers):** software element (e.g. method, variable, class) has too much or too restricted visibility
	Style (S)		**(S.1) - Brackets & Braces:** e.g., single statement after a conditional branch **(S.2) - Indentation:** consistent indentation of the code **(S.3) - Blank Lines:** excess of blank lines or too few blank lines or wrong split of lines **(S.4) - Long Lines:** code statement too long, over a specific amount of characters **(S.5) - Whitespace Usage:** usages of blank spaces in the code **(S.6) - Grouping:** grouping of methods with related functionality or adding class variables at the beginning of the class **(S.7) - Commented out code:** remove code that is commented out (also TODO and FIXME)
	Structure (STR)	**Re-implementation:** Structural defects require an alternative implementation method. For example, replacing the program's array data structure with a vector and knowing the existence of prebuilt functionality that could be used instead of a self-programmed implementation would be considered a solution approach defect. Therefore, solution approach defects are not about re-organizing existing code but rethinking the current solution and implementing it in a different way.	**(STR.1) - Semantic Duplication:** code structures that have a similar intention but are implemented syntactically different **(STR.2) - Semantic Dead Code:** code fragments that are executed, but they do not serve any meaningful purpose and/or have no effect on the result **(STR.3) - Change Function:** change function call to another function because it uses old or deprecated functions **(STR.4) - Standard Coding Conventions:** use exceptions for error messaging instead of return values, use predefined constants instead of magic numbers, built-in data structures instead of own implementation etc. **(STR.5) - New Functionality:** new functionality to ensure evolvability, e.g., create new classes, methods to make code more maintainable **(STR.6) - Strings (Wording):** issues regarding contents of strings, badly composed strings **(STR.7) - Logging:** add the ability to methods for logging results or errors **(STR.8) - Testing:** issues regarding test coverage, wrong tests, additional tests etc.
		Organization: Defects that can be fixed by applying structural modifications to the software. Moving a piece of functionality from module A to module B is a possible strategy for this.	**(STR.9) - Imports:** issues with wrong or missing or unused import statements **(STR.10) - Move Functionality:** move functions, part of functions, or other functional elements to a different class, file, or module **(STR.11) - Long Sub Routine:** split long and complex functions into multiple functions **(STR.12) - Dead Code:** remove code that is never reached and executed **(STR.13) - Duplication / Redundant Code:** remove duplicate code or code that is not used **(STR.14) - Complex Code / Simplification:** restructure or rewrite implementation to make it more understandable **(STR.15) - Statement Issue:** splitting, combining or otherwise reorganizing a statement inside a function **(STR.16) - Consistency:** means the need to keep code consistent in a sense that similar code elements operate in a similar fashion and are more or less symmetrical. For example, similar tasks in similar classes should have similar implementations **(STR.17) - Architectural changes:** code reviews often result in a change to the system architecture, like splitting an interface into two distinct interfaces, introducing abstractions, or the inclusion of design patterns

Table 4.5: CRAM (Code Review chAnges-Model) - Part 2

Activity	Category/Topic	Detailed Change
Functionality & Corrective Maintenance	**Interface (I)**	**(I.1) - Function Call:** call to another part of system or library is incorrect or missing **(I.2) - Parameter:** function call or other interaction has incorrect or missing parameters
	Logic (L)	**(L.1) - Compare:** mistake in a comparison statement **(L.2) - Computation:** computations produce incorrect results **(L.3) - Wrong Location:** correct operation is performed, but it is done too soon or too late **(L.4) - Algorithm/Performance:** inefficient algorithm is used
	Resource (R)	**(R.1) - Variable Initialization:** variables are left uninitialized prior to use. Uninitialized variables may contain any value and using such variable for comparison or calculation produces arbitrary results **(R.2) - Memory Management:** mistake is made in handling the system memory **(R.3) - Data & Resource Manipulation:** defects related to manipulating or releasing data or other resources **(R.4) - Security:** issues related to the application's/software's security aspects **(R.5) - Concurrency:** issues regarding concurrency
	Check (C)	**(C.1) - Check Function:** when a function is called there is also a need to check that the value returned is valid and that no error occurred **(C.2) - Check Variable:** there is a need to check variable **(C.3) - Check User Input:** the need to validate user input
	Larger Defects (LD)	**(LD.1) - Completeness:** partially implemented feature **(LD.2) - GUI:** defects in the user interface code relating to the consistency of the user-interface, and to the options made possible to the user in each situation. **(LD.3) - Check outside code / Domino Effects:** defects that required that part of the application code that was not under review to be checked, as it was likely to contain incorrect code based on the current review.

Table 4.6: CRAM (Code Review chAnges-Model) - Part 3

Artifact	Activity
Other Changes Changes not typically found in source-code files (.java, .py, .cpp etc.) which are nonetheless essential to the runtime of a project	**Commit Message:** updates/changes in the commit message of a submitted patch. Mostly related to wrong description of the change or not capturing all changes
	Continuous Integration / Continuous Deployment configurations: changes to configuration files concerning the Continuous Integration or Continuous Deployment pipeline/setup
	Automated Static Analysis Tools configurations: changes in the configuration of Linters, Checkers, Recommenders used in the project (e.g., Checkstyle, PMD, FindBugs etc.)
	Language or Framework specific: changes to files native to the used programming language. For example MANIFEST for Java
	External Software Documentation: changes to the external Software Documentation files
	Runtime Configurations: docker-configs, ansible playbooks, deployment configs etc
	Other: includes changes to XML, Scripts, README files, HTML files and Version Control

Table 4.7: Distribution of changes in CRAM by *Artifact*

Artifact	# Classified Changes	Percentage
Production Code and Test Code	521	83%
Other Changes	110	17
Total	**631**	**100%**

Table 4.8: Distribution of changes in CRAM by *Production Code and Test Code*

	# Classified Changes	Percentage
Production Code	**440**	**100%**
Functionality	89	20%
Maintainability	351	80%
Test Code	**81**	**100%**
Functionality	15	19%
Maintainability	66	81%

Table 4.9: Distribution of changes in CRAM by *Functionality and Maintainability*

	# Classified Changes	Percentage
Functionality	**104**	**20%**
Check	6	1%
Interface	19	4%
Larger Defects	6	1%
Logic	37	7%
Resource	36	7%
Maintainability	**417**	**80%**
Documentation	128	25%
Structure	252	48%
Style	37	7%
Total	**521**	**100%**

Table 4.10: Distribution of changes in CRAM by *Other Changes*

	# Classified Changes	Percentage
ASAT Configurations	2	2%
Commit Messages	27	25%
Language or Framework specific	57	52%
Other	21	19%
Runtime Configurations	3	3%
Total	**110**	**100%**
% of all Changes	**17%**	

Table 4.11: Distribution of changes in CRAM by *Category*

Category	# Classified Changes	Percentage
Structure	252	48%
Documentation	128	25%
Logic	37	7%
Style	37	7%
Resource	36	7%
Interface	19	4%
Check	6	1%
Larger Defects	6	1%
Total	**521**	**100%**

Table 4.12: Distribution of changes in CRAM by *top 5 Topics*

Topic	# Classified Changes	Percentage (of 521 changes)
Solution Approach	199	38%
Textual Documentation	119	23%
Organization	53	10%
Logic	38	7%
Style	37	7%

Table 4.13: Distribution of changes in CRAM by *top 15 Detailed Changes*

Detailed Change	# Classified Changes	Percentage (of 521 changes)
Strings (Wording)	87	17%
Comments	66	13%
Complex Code/Simplification	36	7%
Testing	27	5%
Duplication/Redundant Code	25	5%
Standard Coding Conventions	24	5%
Data & Resource Management	22	4%
Naming	20	4%
Compare	19	4%
Algorithm/Performance	18	3%
Typos	18	3%
Function Call	17	3%
Logging/Error Handling	17	3%
License Header	15	3%
Commented out Code	14	3%

documentation and code (*e.g.*, naming of classes, methods and variables) of the system (D.2). During the manual classification of changes we found that developers in MCR also carefully review and change or fix the *license headers* (D.3) and fix potential *typos* (D.4) in either in-line comments, Javadocs or Strings in general. Surprisingly, these CR *change types* (D.3-4) were not present in the taxonomy by Beller *et al.*. Related to these changes the study participants claimed that *"tools like PMD, Checkstyle already detect some of such problems (D.4)"*, *e.g.*, typos, *"but are not always so accurate"*. In addition, reviewing and updating the license header of Java classes represents an *"important task to avoid licensing issues"* [68] and to avoid that the software documentation is in general *"not updated or incomplete"*.

Style. Best practices regarding **Style** concern the way code is formatted and appears to developers, *e.g.*, proper code indentation (S.2), the usage of whitespace (S.5), and consistent blank lines (S.3). During the integration of the taxonomy with the one by Beller *et al.*, we noticed that issues regarding the removal of *commented out code* as well as TODO and FIXME comments (S.7) were not considered. Also in this case, study participants claimed that *"tools for this already exists, like PMD and Checkstyles" "but are not always so accurate"*.

Structure. **Structural** defects require alternative implementations and/or refactoring operations in both test and production code. Similar to the taxonomy by Beller *et al.* [15], CRAM differentiates between *re-implementation* and *organizational* changes. *Re-implementation* changes (STR.1-5) involve various activities: the removal or modification of semantic dead code (STR.2) and semantic duplications (STR.1), improvements to the code according to coding conventions (STR.4), the removal of function calls to deprecated functions (STR.3) and creating new functions to ensure the evolvability (STR.5) of the code under review. *Organizational* changes (STR.10-13, STR.15-16) on the other hand are related to defects that can be fixed by applying structural modifications to the software (*e.g.*, refactoring and reorganization of statements inside a method). *Change types* in *re-implementation* and *organizational* changes that were not considered in the schema by Beller *et. al.* are *badly composed strings* (STR.6), *wrong/missing imports* (STR.9), and bad *testing practices*

(STR.8) (*e.g.*, low test coverage, inappropriate tests, the need of additional tests, etc.) In the survey participants mentioned that various changes in MCR are done to enable functions to *log results* (STR.7) or errors. Furthermore, it was also reported that code reviews often result in *architectural changes* (STR.17) to the system. This can involve splitting an interface into two distinct interfaces, introducing abstractions, or the inclusion of design patterns.

Functionality. CRAM shows that during the manual classification of *change types* we noticed that developers in MCR try to address *concurrency* problems (R.5), while in the survey developers strongly highlighted the relevance of (L.4) *performance*, (R.2-3) *resource consumption*, and (R.4) *security* issues (*e.g.*, they claimed that reviewers in MCR should provide answers to questions such as *"have I added a performance bug in my change? - have I added a security bug in my change?"*). This finding is interesting as in previous work security and performance aspects were not considered relevant aspects during code reviews [6]. As reported by a cloud developer in the study, this can be explained by the emerging need to ensure *"the quality of [...] cloud applications, in terms of performance, security and software quality"* in MCR.

> Emerging CR changes and issues occurring in test and production code are related to the need to fix (i) **licensing and security issues**; (ii) **badly composed strings** and **wrong/missing imports**; (iii) **typos** in either in-line comments or Javadocs; (iv) the removal of **commented out code**; (v) the application of **bad testing practices**; and finally, the handling (vi) of **architectural changes** to the system.

Other changes. Changes in non-source-code artifacts reported in Table 4.6 represent a set of CR *change types* that were not present in the schema by Beller *et. al.* [15] and emerged in both the manual classification and taxonomy integration phase. *Change types* in this category are mostly related to the configuration of continuous delivery (CD) and continuous integration (CI) files (O.2), ASATs (O.3) and runtime files (O.6). Furthermore, this category captures *change types* in commit messages (O.1), external documentation (O.5), issues/changes in framework or language specific files (O.4),*e.g.*, *pom.xml* for Maven[2] projects, and other artifacts (O.7), *e.g.*, scripts and README files. Changes in CI/CD files are most commonly performed to fix suboptimal instantiations of delivery pipelines. ASATs configurations are changed in order to improve their effectiveness and performance (PMD, Checkstyle etc.) and changes to runtime files ensure that the project will be correctly built (*e.g.*, Docker[3] configurations).

These findings are particularly interesting, as differently from previous research [6], reviewers in MCR also focus on CD and CI topics and practices, something that needs to be further investigated in future research.

> Emerging CR changes related to non-source-code artifacts concern **configuration in CD and CI** files, changes in **runtime configurations**, **ASATs configuration** files and other non-source-code artifacts (*e.g.*, commit messages and external software documentation).

[2]https://maven.apache.org/
[3]https://www.docker.com/

Emerging technologies. In summary, our findings clearly show that most of the novel CR *change types* are related to changes and issues that developers perform or have to deal with because of the availability of emerging development technologies (*e.g.*, cloud-based technologies) and practices (*e.g.*, CD/CI). Especially the management of CD/CI pipelines [26, 27] and static analysis tools configuration [8, 55, 67] are practices that improve both developer productivity and the development process of modern software systems as a whole [8, 38, 63, 67]. The growing application of these practices has pushed developers to perform additional activities or tasks during code reviews, specifically with the aim of reviewing, re-thinking, and changing software artifacts that impact the CD and CI processes as well as the effectiveness/performance of static analysis tools.

> Most of novel CR *change types* in CRAM are related to changes or issues that developers perform or have to deal with because of the availability of ***emerging development technologies*** (*e.g.*, cloud-based technologies) ***and practices*** (*e.g.*, Continuous Delivery and Continuous Integration).

Distribution of change types. Our findings are further supported by the quantitative analysis presented in Section 4.1, where 631 review comments and their related changes were classified into CRAM. The results in Table 4.7 show that about 83% of changes occur in either production or test code and 17% are related to changes in non-source-code artifacts. Moreover, when looking only at changes in production and test code (Table 4.8), around 80% of changes are performed in maintainability categories, whereas the remaining 20% involve functionality changes. Finally, as also confirmed by participants in the study, around half (48%) of all changes are related to the structure of the code and 25% constitute changes in the documentation (Table 4.9 and 4.11). When looking at the tables it is evident that most changes are related to maintainability issues. As investigated in previous work, it was found that around 75% of all fixed issues in MCR are non-functionality related [15, 44]. This thesis further supports this finding.

> More than **80% of all changes in MCR are related to maintainability issues**, supporting the findings of previous works [15, 44]. Furthermore, emerging technologies and practices push developers to perform additional changes and tasks in MCR. Around 17% of changes are performed in non-source-code files such as CD/CI and ASATs configurations.

4.2 Towards the Automation of MCR

While Section 4.1 presented results and findings regarding recurring issues that developers perform in MCR, the following section shows results and findings gathered from the survey evaluation regarding *automation needs*. More specifically, we discuss the results aimed at understanding which tools and solutions developers would need during code reviews, with a particular focus on the recurrent and critical changes investigated in the previous section, and how developers would approach the automatic detection and fixing of required code review *change types* in order to improve a submitted patch to code review.

Applying the methodology described in Chapter 3, the survey feedback (Q2.1-Q2.7) was carefully analyzed, encoded and grouped into clusters of *recommendations*, *data specifications* and *techniques*. The following tables present the qualitative results of the feedbacks given in the survey,

Table 4.14: Q1.3: Distribution of recurrent change topics

Sub-category	Count	%
Structure	*126*	***48.4%***
Documentation	*64*	***24.6%***
Logic	18	7.1%
Style	19	7.1%
Resource	17	6.7%
Interface	9	3.6%
Check	3	1.2%
Larger Defects	3	1.2%
Resource	1	0.2%
Total	**260**	**100%**

Table 4.15: Q1.4: Expected feedback in MCR

Category	Ranking	Sub-categories
Documentation	14 Cases (11.4%)	no sub-category
Functionality	45 Cases (36.6%)	Check(3), Completeness (2), Data & Resource Manipulation (3), Interface (4), Large Defects (3), Logic (12), Performance (7), Resource (7), Security (4)
Other	7 Cases (5.69%)	no subcategory
Other Changes	9 Cases (7.3%)	Automated Static Analysis Tools configurations (3), Continuous Integration/Continuous Delivery configurations (2), Runtime Configurations (2), no subcategory (2)
Structure	34 Cases (27.6%)	Architectural Changes (5), Complex Code (7), Logging (1), Duplication (1), Standard Coding Conventions (2), Testing (6), no subcategory (12)
Style	14 Cases (11.4%)	no subcategory
Total	**123 Cases (100%)**	

focusing on required automation that, from a *developer point of view*, are still needed.

Recurrent CR change types. Table 4.14 reports the changes that participants consider the most recurring in CR, whereas Table 4.15 reports the feedback developers would like to receive from reviewers and Table 4.16 summarizes the feedback they actually receive in code reviews. By looking at Table 4.14 it can be seen that, according to the study participants, the most prominent *change types* occurring in code reviews are related to the structure (48.4%), e.g., refactorings, and reorganizations of test and production code and the software documentation (24.6%). Other CR changes types (e.g., changes in the logic and the style of the code of the patch under review) rarely occur, each covering less than 8% of the code review topics, and all together correspond to around 27% of the total CR changes performed in a patch.

Envisioned approaches. The survey participants provided more than 400 comments on automation needs (Q2.1-Q2.7) characterizing MCR. In Table 4.17 the solutions and approaches which were proposed the most often by the developers are summarized, with a particular focus on the new CR *change types* that emerged in the empirical investigation of RQ_1. The proposed solutions

Table 4.16: Q1.5: Feedback received in MCR

Category	Ranking	Sub-categories
Documentation	20 Cases (21.1%)	Naming (1), Typos (1), no category (18)
Functionality	26 Cases (27.4%)	Check (2), Interface (1), Larger Defects (4), Logic (11), Performance (2), Resources (2), Security (3), no category (1)
Other	4 Cases (4.2%)	-
Other Changes	1 Case (1.1%)	-
Structure	28 Cases (29.5%)	Architectural Changes (4), Complex Code (3), Duplication (1), Standard Coding Conventions (3), Testing (4), no category (13)
Style	16 Cases (16.8%)	no category
Total	**95 Cases (100%)**	**100%**

in column three of Table 4.17 are clustered into abstracted solutions, as developers often referred to similar types of automated solutions.

4.2.1 Emerging Automation Needs

In the following we discuss our results and answer RQ_2: *"What are the emerging automation needs of developers in MCR?"*. In a first step, by following the approach shown in Figure 4.1, the needed automation in MCR as evaluated in the survey questions Q2.1-Q2.5 are discussed and our findings presented. Then, we provide a general overview over the recommendations, techniques and data that developers proposed in Q2.6 and Q2.7 followed by promising examples of how specific automation can be implemented for the automation needs mentioned by the developers.

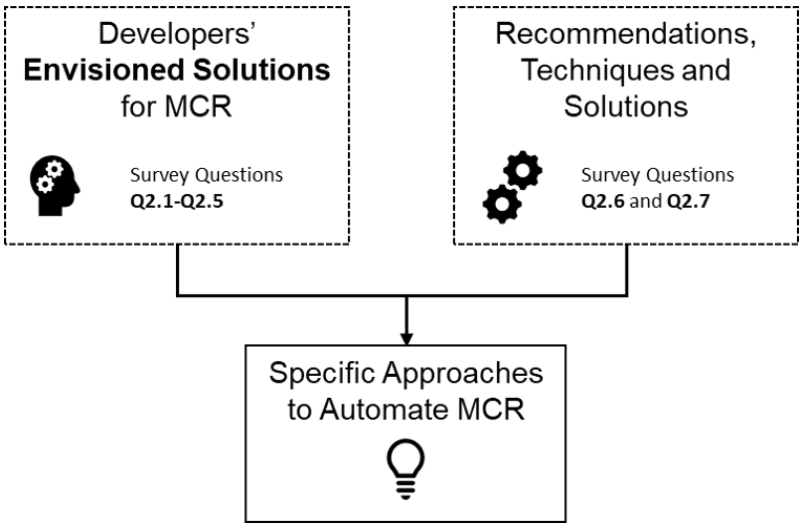

Figure 4.1: Approach for Evaluating Developers' Automation Needs

Table 4.17: RQ_2: Developers' Envisioned Solutions.

Category	Detailed Change	Abstracted Solution
Documentation (56)	- General (32) - Comments (3)	31 - **Automatically detecting and fixing** documentation issues (documentation incomplete or inconsistent with the source code) 4 - **Generation** and replacement of inconsistent documentation/comments
	- Naming (12)	12 - **Renaming suggestions** based on standard naming used in the codebase
	- Typos (5)	5 - **Automatic** spell checking (also grammar) and fixing
	- License Header (1)	1 - **Generating** License Header
Style (40)		27 - **Evaluate Style Consistency** with the style adapted by the team and auto-fix the style issues 13 - **Use existing tools** for these issues, e.g., PMD and CheckStyle
Structure (29)	- Refactoring (8) - Duplicated, (Semantic) Dead, Unused, and Deprecated Code (19) - Architecture violations (2)	19 - **Detection of duplicated**, unused, (semantic) dead, and deprecated code 8 - **Refactoring suggestions** for test and production code 2 - **Detect architectural violations**
Functionality (19)	- Performance (4) - Resource (11) - Security (4)	12 - **Auto-fix** of performance, resource issues 5 - **Detect security issues** 3 - **Performance and resource analysis**
Other Changes (9)	- CD/CI configurations (4) - SATs configurations (2) - Runtime configurations (3)	4 - **Recommend/improve CD/CI** configurations 3 - **Recommend/improve runtime** configurations 2 - **Recommend/improve SATs** configurations

Envisioned Solutions

Expectations of developers towards MCR. As expected, the results in Table 4.16 highlight how the feedback developers receive from reviewers is highly consistent with the changes they actually perform (Table 4.14). However, when looking at both Table 4.15 and Table 4.16, it is evident that the feedback developers receive in MCR is often not satisfactory and rarely meets all the current expectations of developers. One participant mentioned that *"many of the problems we face during code review are related to the miss-match between expectations and outcomes of a code review [...] reviewers provide feedback that are not exhaustive or timely reported. This often makes code reviews unproductive"*. It is interesting to see that feedback on the structure of the code and documentation aspects have a lower representation in the expected feedback (4.15) in comparison to what feedback is actually received in MCR (4.16), while comments related to the categories *Functionality* (*e.g.*, performance and resources) and *Other Changes* are more important nowadays. For instance, 8% of the participants in the study stated that they would like to receive comments related to CD/CI and SATs configurations, while only 1% of them receive such feedback. This general finding highlights the gap between the expectations developers have towards the code review process and the outcome of these reviews. Furthermore, this emphasizes our previous findings that emerging technologies and practices have to be considered more strongly by reviewers in MCR to fulfill developers' expectations.

> Problems in MCR are often related to a **mismatch between expectations and outcomes of a review**. Emerging technologies and practices require more exhaustive and detailed feedback during reviews. More specifically, emerging technologies (cloud-based technologies) and practices (CD/CI) have to be considered more strongly by reviewers in MCR to fulfill developers' expectations.

The most frequent CR categories for which further automated approaches would be needed (Table 4.17) are *Documentation* (56), *Style* (40), *Structure* (29), *Functionality* (19), and *Other Changes* (9).

Documentation and style solutions. For the Documentation category we received 56 distinct feedbacks by developers in the study. To automate Documentation issues in MCR, developers believe that advanced automation need to be conceived that are able to ***detect and fix issues related to the incomplete or inconsistent documentation*** with respect to the source code. Recent work in this area conducted and explored this problem [75] but not in the context of code review. Further feedbacks by participants revealed that approaches are required able to ***directly generate the required documentation and comments*** including the **license header**. There is also a demand for spell-checking, integrated into the code review process to find potential typos in the Documentation and Strings in general. Moreover, the evaluation showed that developers expect more sophisticated recommender systems for MCR in both the Documentation and Style categories. One recurring issue is the problem of naming. Here, developers expect solutions that detect wrong naming and provide ***renaming recommendations*** according to the naming conventions and policies of the project. When asked about Style related issues, most developers stated that they would expect the ***detection and fixing of coding style errors*** in MCR. However, many developers also reported that tools like Checkstyle could or should already be sufficient to handle some of the style related issues.

In addition to solutions that directly **generate the required documentation and comments** in software artifacts and **detect and fix licensing issues**, automation regarding **naming** are needed by developers in MCR. Not only should issues in the Naming category be detected, but tools should recommend appropriate fixes by considering naming conventions and policies of the project, ensuring the consistency with the existing code base.

Structural solutions. According to developers in our study, various solutions regarding structural CR changes are desired, foremost ***refactoring recommendations for both test and production code***. For instance, a developer mentioned the need for refactoring recommendations *"of tests not based only on coupling concepts but also encapsulating the need of having explicitly separated testing performance from functional testing"* in order to facilitate, for example, tests of *"different properties of micro-services of a cloud application"*. Furthermore, participants mentioned the need to have timely feedback about *"test/code smells (bad design choices) added"* in the patch under review, *e.g.*, provide ***"auto-generated feedback based on test/code smells notions, providing an overview on overall test/code quality and readability"***. The study also shows that automated tools are needed for the detection of duplicated, unused, (semantically) dead code. One participant mentioned: *"highlight dead code, unreachable code, and suggest refactoring options [...] for duplicates"*. Other participants stated that having tools which can find deprecated and architectural violations, answering questions such as *"did I have introduced imperfections at the level of Architecture?"*, would be beneficial to the MCR process.

Developers need automation solutions for structural code issues in MCR such as (i) **refactoring suggestions**; (ii) automated **feedback on test and code smells** by providing an overview over the quality and readability of the code; and (iii) automation that **detect and highlight duplicated, unused and dead code** in a submitted patch.

Functionality solutions. As can be seen in Table 4.17, there is a substantial demand from developers for tools that are able to detect ***performance, resource consumption and security issues***. For instance, a participant of our study reported: *"... a company producing self-driving cars, in [...] code review will require also to observe potential security and or testing issues"*. Participants also stated that *"Performance and Security issues are the more difficult to automate..."*.

Important automation requested by developers are solutions able to detect **performance, resource and security** issues.

Solutions for emerging technologies and practices. From the results of RQ_1 we can see that most CR *change types* that were added to `CRAM` in 4.1 are related to changes or issues that developers perform or have to deal with due to emerging development technologies and practices. This also influences the type of solution developers would need in the future. Participants highlighted that tools ***recommending, improving, monitoring*** CD/CI, runtime and SATs configurations are desired. One participant mentioned that *"automation should consider more recent concept of CD, CI, [...] thus providing feedback on: the way a CD/CI pipeline is structured or is efficient..."*.

Table 4.18: RQ_2: Developers' Recommendations.

	Recommendations		
Taxonomy high-level	Learn from past data (code review changes)	Find patterns (antipatterns)	Check against codebase
#mentions by participants	**57**	**60**	**5**
Documentation	0	7	3
Functionality	18	19	0
General Approaches	14	5	0
Other Changes	3	2	0
Structure	14	16	1
Style	8	11	1

Developers require more sophisticated tools supporting them in MCR. Specifically, tools are needed that are able to give **feedback on the efficiency of CD/CI pipelines** as well as **suggest the right configurations** for these emerging practices.

Enhancing MCR - Recommendations, Techniques, and Data

Table 4.18, 4.19, and 4.20 summarize the recommendations, techniques and data found in our study evaluation related to possible ways how certain MCR activities can be supported or automated. Table 4.18 shows the methodology to apply, Table 4.19 highlights what specific techniques should be used, and Table 4.20 provides information about what type of data has to be studied according to the developers in our survey.

Recommendations. As can be seen in Table 4.18, the developers generally suggest the notion of patterns and anti-patterns (60 mentions by participants) to detect issues in all categories of CRAM, especially for Functionality (19) and Structure (16) issues, with the goal to generate templates of well and badly written code, which can then be matched against newly submitted code. This would allow to determine if the introduced change contains issues or not. As an example, one developer provided the following feedback: *"I would try to understand which problems are recurrent in code reviews, e.g., defining patterns and anti-patterns characterizing the 'code review process and collaborations'. Then I would automatically leverage tools for detecting such anti-patterns to highlight potential problems."*. Furthermore, many developers suggested learning from past data (57), *e.g.,* study the history of changes in the software repository, to gain more insights into how teams develop code and what recurrent issues they fix in MCR: *"[...]study previous changes in the software and its documentation to learn more about upcoming (and recurrent) mistakes. I would mine this information with some data mining techniques and machine learning to learn these patterns and/or identify some of the problems."*. This recommendation was proposed most often for the Functionality (18), Structure (14), General Approaches (14) and Style (8) categories in CRAM.

The most mentioned recommendations by developers are the (i) notion of defining and finding **patterns/anti-patterns** as well as (ii) analyzing the **history of changes in the software repositories** to detect and fix recurring issues in MCR.

Table 4.19: RQ_2: Developers' Techniques.

	Techniques											
Taxonomy high-level	Machine Learning (predictions)	NLP (Text Mining)	Data Mining	Static Code Analysis	Dynamic Code Analysis	Summarization Techniques	Regex parsing	Manual Analysis	Literature (state of the art)	Integrate into IDE	Use existing Tools	Rely on Compiler
#mentions by participants	**28**	**19**	**12**	**27**	**15**	**7**	**3**	**1**	**0**	**6**	**14**	**3**
Documentation	4	4	1	3	1	0	1	0	0	2	3	0
Functionality	8	7	3	8	7	5	0	0	0	1	0	3
General Approaches	6	3	3	8	4	1	0	0	0	1	0	0
Other Changes	1	0	0	0	0	0	0	0	0	0	0	0
Structure	5	2	3	5	2	1	0	0	0	1	2	0
Style	4	3	2	3	1	0	2	1	0	1	9	0

Techniques. Table 4.19 reports developers' opinions of which techniques should be applied to different categories of CRAM. The technique mentioned the most is **Machine Learning** with a total of 28 mentions, while the use of **Static Code Analysis** was mentioned the second most frequent with 27 cases. With 19 mentions, the use of **NLP** techniques is also thought to be a viable technique in the automation of MCR. Moreover, some developers believe that issues such as Documentation (3) and Style (9) are already partly handled by existing tools (14) and do not need to be further automated.
When looking at Machine Learning, developers in the survey believe its application is most suited for issues regarding Functionality (8), General Approaches (6) and Structural issues (5). Techniques involving Static Code Analysis were most prominent for the Functionality (8) and General Approaches (8) categories, while NLP was suggested foremost for Functionality (7 cases), Documentation (4 cases) and Style (3). As mentioned by a participant in the study: *"i would use NLP analysis to study the coding style of the team/project, thus detect recurrent style issues (explaining what is wrong in my style)..."*. It is interesting that for almost every high-level CRAM category, participants stated that *change types* could be detected and fixed by a learning approach. Also for emerging practices such as CI/CD and cloud-based technologies, participants suggest using learning approaches to provide support with recurrent issues in configuration files.

> The most promising techniques suggested by developers are **Machine Learning** for Functionality, Structural and General issues; **Static Code Analysis** for Functionality and Other issues in CRAM and **Natural Language Processing** for Functionality and Documentation issues in MCR.

Data. Table 4.20 reports developers' opinions of what type of data should be studied in the context of different CRAM categories in order to provide useful information for the envisioned

Table 4.20: RQ_2: Developers' Data.

	Data					
Taxonomy high-level	Metrics (in general)	Change metrics	Code metrics	OO-metrics	Natural language	Code documentation
#mentions by participants	**32**	**15**	**6**	**1**	**0**	**10**
Documentation	5	1	0	0	0	2
Functionality	13	3	0	0	0	3
General Approaches	4	6	2	0	0	4
Other Changes	2	0	0	0	0	0
Structure	4	3	3	1	0	0
Style	4	2	1	0	0	1

automation approaches. According to the participants in our study, metrics are very important when conceiving tools to automate and support MCR: *"I would define lightweight (not expensive metrics to compute) metrics to measure of detect the issues."*, *"From one side code and code changes metrics + code change patters would a useful information to study..."* or *"For each problem/issues, I would focus on recurrent and relevant metrics, patterns and anti-patters"*. Out of 64 mentions in the *data* section, 54 (84%) suggest that metrics should be computed, studied and applied to fix and detect recurrent issues in MCR. As can be seen in 4.20, the *General Approaches* category which comprises solutions and data recommended not specifically for only one `CRAM` category, developers highly recommend the use of **change metrics**, **code metrics** and **object-oriented metrics** (OO-metrics). The remaining 10 (16%) mention that the **documentation** of the software project should be analyzed, comparing and studying previous changes to learn more about potential upcoming issues, as mentioned by one developer it would provide *"useful information to study previous changes in the software and its documentation to learn more about upcoming (and recurrent) mistakes."*

> According to developers, important data for the automation of MCR are metrics. Around **84%** of mentions regarding suggested data to study are related to metrics such as **change metrics**, **code metrics** and **object-oriented metrics**. Other mentions (16%) concern the analysis of code documentation.

Enhancing MCR - Specific Approaches to automate MCR

This subsection discusses specific automation approaches for the categories of `CRAM` as well as approaches targeting overall improvements to MCR. To that extent, the feedback provided by the developers in our survey (Q2.6 and Q2.7) was carefully analyzed, labeled and clustered. The encoded survey feedback is available in the appendix. By considering the insights found regarding the various required solutions (Subsection 4.2.1) and the proposed recommendations, techniques and data in Subsection 4.2.1, we propose specific combined approaches how activities of MCR can be automated. Furthermore, we state how it can enhance and facilitates review activities

performed by developers.

Specific approaches for Documentation. Our findings in subsection 4.2.1 show that developers are interested in solutions that can (i) generate the required documentation; (ii) detect and fix licensing issues; and (iii) support developers with naming issues. Furthermore, in the evaluation in subsection 4.2.1 we found that developers have the opinion that the automation of the Documentation category is best approached by studying patterns/anti-pattern as well as historic changes with NLP by defining and using specific documentation metrics. In the following we provide the most promising and needed automation solutions for Documentation, distilled from the evaluation of the survey and our findings in the previous subsections.
Generate Documentation: (1) As mentioned by one participant, a promising solution would be to *"Try to find characteristics e.g. for a method description based on the code itself"* and then use the *"method name + parameters + other method and classes called within the method + instantiation + ... to predict what that method actually does..."* and automatically generate documentation out of it.
(2) Define patterns/anti-patterns and detect metrics by studying the history of the project in relation to Documentation issues to predict required changes or recommending solutions: *"I would try to analyze code and other changes from the history of the project, detecting, metrics patterns, anti-patterns, thus predicting required changes, or recommending solution when possible."*
(3) Use existing tools (Checkstyle and other Linters) to fix issues in Documentation, or combining the tools and integrate them into the review process.
Support for naming issues: In order to suggest and fix potential issues in Naming *"compare variables, statements and function calls not only against syntax definitions but against similar variables, statements and function calls from the codebase being changed, enabling a developer to adapt to local naming customs, and to see (and match) existing examples of the code being reviewed"*.

Specific approaches for Style. Our findings in subsection 4.2.1 show that developers are interested in solutions that can check the style consistency of a change with the style adapted by the team and provide fixes for these issues. In the evaluation in subsection 4.2.1 we found that developers have the opinion that the automation of the Style category is best approached by studying patterns/anti-pattern as well as historic changes with NLP by defining and computing metrics. In the following we provide promising and needed automation solutions for Style issues.
Evaluate style consistency: Use Machine Learning and NLP to detect anti-patterns and recurring issues related to Style. By learning these recurring issues, required changes in new submitted patches can be detected.
Use existing tools: Many developers think that Style issues are already handled by current tools such as FindBugs, Checkstyle, PMD, and instead suggest the integration of these tools into the review process: *"There are many code checkers out there (FindBugs, ScalaStyle, PMD, ...). Maybe implement an integration with one of these tools during the reviewing process."*. Additionally, the removal of certain warning outputs by these tools can be enforced, thus reducing review time, as stated in the work by Panichella *et al.* [55].

Specific approaches for Structure. Developers stated that they would need the following automations in relation to Structure issues: (i) refactoring suggestions of test and production code; (ii) automated feedback on test and code smells by providing an overview over the quality and readability of the code; and (iii) automation that can detect and highlight duplicated, unused and dead code. To approach the automated fixing of these needs, developers suggested using Machine Learning techniques to learn from historic changes to the software as well as define and find patterns/anti-patterns.
Refactoring suggestions: (1) Detect patterns by comparing new changes to existing code, thus recognizing if a better design pattern could be applied: *"Pattern detection with suggestion ->e.g. I*

see you build something which would be smart to structure in a factory pattern"
(2) Study code, change metrics and change patterns as well as the software documentation and historic changes, and then use this information with Machine Learning techniques to learn about recurrent and upcoming issues in MCR to predict and identify refactoring possibilities.
Feedback on code smells in the test and production code : Similarly to the approach mentioned for refactoring suggestions, use a Machine Learning approach to learn about recurrent issues regarding changes in the test and production code to predict and identify smells.
Duplicated, unused and dead code: Use Static Code Analysis and pattern/template-matching to evaluate the correctness of submitted code, although *"not all the duplication should be refactored or merged, which brings the challenge for research to locate the merge/refactoring-oriented code duplication."*. This is not an easy task as mentioned by a developer: *"If a function is duplicated between two classes, that may be intentional to avoid coupling those classes, so we might want to ignore it. But if a function is duplicated within two classes that both inherit from the same parent class than the solution would be to put that function in the parent class."*

Specific approaches for Functionality. Automation solutions are needed for (i) performance; (ii) resource consumption; and (iii) security issues. From the evaluation of the survey and our findings in previous sections we can state three recurrent approaches that developers suggest for these issues.
Build integration with contemporary tools: *"These kind of tools already exist, but are not necessarily integrated with the commonly used IDEs (e.g. valgrind for memory checks)"*. By combining existing tools into the review process, specific issues related to Functionality can be detected and fixed.
Utilize patterns and anti-patterns: This approach involves the definition of patterns and anti-patterns to detect recurrent issues in the functionality of the code as stated by one developer: *"I would use the notion of patterns, antipatterns to also fix/handle problems on resource consumption of my code, something not available in code review practices."*
Learning approach: Use of code and change metrics as well as historic data analysis in combination with Machine Learning and data mining techniques to detect defects and predict needed changes in a patch: *"I would try to analyze code and other changes from the history of the project, detecting metrics patterns, anti-patterns, thus predicting required changes, or recommending solution when possible."*.

Specific approaches for emerging issues (Other Changes). Particular solutions are requested for recommending, improving and monitoring CD/CI, runtime and ASATs configurations. Similarly to Functionality, our findings show that developers believe this is best approached with (i) studying patterns/anti-patterns and metrics characterizing non-source code artifacts from historical data, then (ii) observing these anti-patterns in the development process and practices (*e.g.*, trends in change and code metrics, build failures, etc.) with data mining and Machine Learning techniques, and then (iii) leveraging NLP and summarization techniques [35, 48, 54] to provide more context about the detected issues, and recommending changes to fix the patch.

Summary. As our evaluation showed in Section 4.1, Documentation issues represent around 25% of all issues that were classified into `CRAM` providing evidence that the automated fixing of these issues would largely benefit MCR. Not only would it reduce the review time, but also keep the project consistent with either documentation policies and the naming conventions applied by the team. This is also the case for Structure issues (48%); the automated fixing and detection of such issues would greatly increase the productivity of developers in code review. Furthermore, for Style (7%) issues, considering the feedback received in the survey, these *change types* might indeed already be handled well by current approaches and tools. Nonetheless, the occurrence of

these issues in reviews can potentially be further reduced by analyzing which ASATs warnings are most needed to fix and at the same time by enforcing the fixing of these warnings. The same is true for issues regarding emerging technologies and practices (category Other Changes) as well as for Functionality issues, where learning about patterns and metrics can potentially detect many recurring issues and support developers in their reviewing activities.

> The most mentioned approaches by developers can be broken down into the following steps: (i) perform a manual analysis to investigate **patterns**, **anti-patterns**, **change metrics** and **documentation metrics**; (ii) then leverage **NLP** or **machine learning** techniques in combination with **static code analysis**; (iii) to characterize and **predict** further changes or **detect** issues in a submitted patch.

Moreover, it is interesting to observe for all categories in Table 4.19 how none of the participants mentioned the possibility to use an existing technique from the literature, but rather implement solutions based on customized approaches leveraging machine learning, NLP and data mining techniques to modeling recurrent issues with the notion of anti-patterns, and change metrics. Furthermore, we found that many developers require solutions that are tailored to their specific development team. As mentioned by one participant *"I think a tool that can automatically recognize team "culture" and detect deviation from it, would be helpful in the reviewing process."*.

4.3 Enabling Automation: a Proof of Concept

A *proof of concept* towards the envisioned automation in MCR considering developer needs was developed by leveraging suggested code metrics and other features with the intention to use these characteristics in a Machine Learning approach.

4.3.1 Metrics

Motivation. As can be seen in our findings in the last subsection regarding the proposed approaches to automate certain aspects of MCR, metrics in combination with Machine Learning techniques are considered the most promising solutions towards the automation of MCR. Multiple feedbacks in our survey motivate the use of metrics, such as *"[...]detection and auto-fix are still not available for code review[...] Metrics based on static, dynamic analysis or detection strategies based on some nlp analysis and similar approaches would be useful here I guess."* and *"From one side code and code changes metrics + code change patters would a useful information to study [...]"*. Furthermore, one participant mentioned that she believes the *"[...] best idea is to have a look at changes to heterogeneous types of data and recurrent change patterns to provide the automation [...]"*. Furthermore, many developers believe that analyzing the history of code changes gives information about upcoming or recurring issues. More specifically developers suggested the use of (i) code metrics; (ii) change metrics; (iii) object-oriented features to detect refactoring possibilities; (iv) integration of existing ASATs into the review process; (v) NLP features; and (vi) detection of patterns/anti-patterns on source code level.

To that end, the proposed metrics are presented and an implementation approach to extract these metrics is suggested. Furthermore, we highlight for each metric what type of information it could provide and towards which elements in `CRAM` this information could be used.

Static Code Metrics. One set of metrics are the static code metrics that can be extracted from the source code. Common are metrics revolving around Lines of Code (LOC) and cyclomatic complexity. LOC metrics consider the Source Lines of Code (SLOC) and various metrics can be computed such as Pyhsical LOC (SLOC-P), Blank Lines of Code (BLOC), Comment Lines of Code (CLOC) and Logical Lines of Code (SLOC-L) [50]. Another common metric is the Cyclomatic Complexity Number or McGabe metric (CNN) [45], which computes the number of independent logical decision paths in a program structure.

These metrics provide valuable information about the general structure of the source code and its complexity, meaning that substantial differences in these metrics could increases the potential for source code containing issues and bugs. As researched in [10, 77], static code metrics were successfully used to predict if a file contains defects. Similarly, the use of these metrics in MCR could provide a first step towards the detection of issues or bugs in newly submitted patches. According to our findings in the previous section and the feedback by developers, static code metrics can be particularly useful for *Functionality* issues in the code.

In an implementation, the available tool Unified Code Count[4] (UCC) can be used to compute static code metrics during the extraction process between two patches in Gerrit Code Review.

Table 4.21: Change Metrics

Metric Abbreviation	Description
AGE	Age of a file in years (counting backwards from current revision)
CODE_CHURN	Added lines of code minus deleted lines of code of current patch/commit
CODE_CHURN_AVG	Average of added lines of code minus deleted lines of code over all revisions
CODE_CHURN_MAX	Maximum of added lines of code minus deleted lines of code over all revisions
CODE_CHURN_MIN	Minimum of added lines of code minus deleted lines of code over all revisions
CODE_CHURN_TOT	Total of added lines of code minus deleted lines of code over all revisions
LOC_ADD	Lines of code added to a file of current patch/commit
LOC_ADD_AVG	Average over all revisions of the lines of code added to a file
LOC_ADD_MAX	Maximum over all revisions of the lines of code added to a file
LOC_ADD_MIN	Minimum over all revisions of the lines of code added to a file
LOC_ADD_TOT	Total over all revisions of the lines of code added to a file
LOC_DEL	Lines of code deleted from a file of current patch/commit
LOC_DEL_AVG	Average over all revisions of the lines of code deleted from a file
LOC_DEL_MAX	Maximum over all revisions of the lines of code deleted from a file
LOC_DEL_MIN	Minimum over all revisions of the lines of code deleted from a file
LOC_DEL_TOT	Total over all revisions of the lines of code deleted from a file
NAUTH	Number of distinct authors that checked a file into the repository
NREV	Number of Revisions of a file
PATCH_LINES_ADD	Number of lines added in current patch
PATCH_LINES_DEL	Number of lines deleted in current patch

Change Metrics. Previous research [49,59] showed that in contrast to static code metrics, change metrics sometimes offer more useful information when used for defect prediction. Furthermore, participants highlighted, that such metrics could be very useful in the context of automating MCR: *"From one side code and code changes metrics + code change patters would a useful information [...]"* for detecting and fixing recurrent issues. These metrics provide information about the process of code review such as how many authors introduced changes in a particular file, the age of a file and how many lines were deleted or added. This provides valuable information in the sense

[4] http://csse.usc.edu/ucc_new/wordpress/

Table 4.22: CK-Metrics

Metric Abbreviation	Description
CBO	Coupling Between Objects
DIT	Depth of Inheritance Tree
LCOM	Lack of Cohesion in Methods
LOC	Source Lines of Code
NOC	Number of Children
NOF	Number of Fields
NOM	Number of Methods
NOPF	Number of Public Fields
NOPM	Number of Public Methods
NOSF	Number of Static Fields
NOSI	Number of Static Invocations
NOSM	Number of Static Methods
RFC	Response For Class

that files that undergo many revisions or need to be changed by multiple authors might have a higher tendency to still contain issues. According to the findings in the previous section (Table 4.20), change metrics can provide the most useful information when used to fix *Structure* and *Functionality* issues or defects in configuration files (*Other Changes*).

Change metrics are extracted from the repository and can have various characteristics. For the purpose of the *proof of concept* the change metrics in Table 4.21 were considered.

CK Metrics. The projects analyzed in this thesis are all written in Java and thus, they are object oriented implementations. Moreover, participants in the study are interested in solutions towards the detection of possible refactoring issues (Architectural Changes). Therefore, to obtain more characteristics between specific patches in CR, object oriented metrics can be extracted from the source code. Considered for this *proof of concept* are metrics based on the metrics suite by Chidamber-Kemerer (CK) [22], which compute metrics such as the Weighted Methods per Class (WMC) and Depth of Inheritance Tree (DIT). The full list of considered metrics can be found in Table 4.22.

Metrics such as Coupling Between Objects (CBO) give insights in how well the software's architecture is designed, meaning that classes with higher CBO might be more prone to errors and need possible refactoring. Another important metric is Depth of Inheritance Tree (DIT), that provides information about "the maximum length from the node to the root of the tree" [22], suggesting that classes with a higher DIT (i) inherit more methods, therefore making it more difficult to predict their behavior; (ii) involve a higher design complexity since more methods and classes are involved; and (iii) have a higher potential for reusing inherited methods. While (i) and (ii) suggest that a higher DIT number is not desirable, (iii) indicates that entities lower in the inheritance trees make more use of existing methods, and thus, make the code easier to maintain.

By computing CK metrics on a newly submitted patch and comparing the results to the values in the previous version of the patch, potential errors and patterns can be found. An existing tool[5] can be considered to compute relevant metrics between two patches.

ASAT Warnings Metrics. As mentioned in Related Work (Chapter 2), Automated Static Code Analysis tools (ASAT) such as PMD, FindBugs and Checkstyle are popular tools to either find

[5]https://github.com/mauricioaniche/ck

style related issues or structural defects in the source code. A study conducted by Panichella *et al.* [55] analyzing the removal rate of PMD and Checkstyle warnings in open-source projects, found that certain warnings are removed more often than others and that some defects are discovered by ASATs. Moreover, Beller *et al.* [16] introduced a mapping of 1'825 ASAT-specific warnings to a General Defect Classification (GDC) scheme consisting of 16 top-level classes (Figure 4.2). As can be seen, the top-level classes in the GDC are similar to the categories comprising CRAM, such as Check, Concurrency, Interface, Logic, Resource, Documentation, Naming, Redundancies, Simplifications and Style Conventions. Furthermore, we found in our study that various participants advised using ASAT tools related to certain issues in the categories Documentation and Style: *"There are many code checkers out there (FindBugs, ScalaStyle, PMD, ...). Maybe implement an integration with one of these tools during the reviewing process."*.

For the purpose of this *proof of concept* warnings from PMD and Checkstyle are considered, specifically, how the number of warnings in each top-level class in the GCD change between different patches.

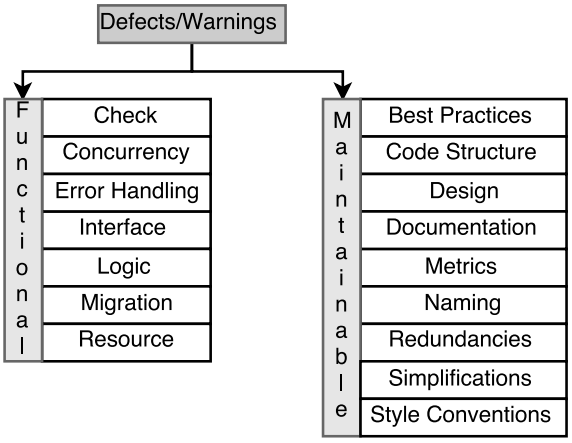

Figure 4.2: The General Defect Classification (GDC) by Beller *et al.* [16].

Commit Messages. As recent work has shown, NLP can successfully be used to classify natural language texts into topics and clusters of similar subjects [25, 56]. Therefore, the commit message of the patch can be analyzed. Words in the commit message are then weighted using the *tf-idf* score [7] opposed to simple frequency counts, as it assigns a higher value to rare words (or group of words) appearing in the message, and a lower value to common words. This allows to identify the most important words in the commit message. The usage of commit messages is supported by the feedback given in the survey where participants suggested the usage of natural language processing (NLP) to detect characteristics of patches such as the following feedback by one of the participants: *"i would use NLP parsing to detect recurrent issues [...]"*. NLP features are useful to detect potential recurring issues in Documentation, Functionality, Structure and Style.

CHANGEDISTILLER Change Types. In order to obtain detailed features describing the source code on the syntax level, CHANGEDISTILLER [31] can be used. The tool finds fine-grained source code changes using tree differencing and basic tree edit operations (insert, delete, move) between two syntax trees by transforming one into the other. CHANGEDISTILLER was successfully applied in predicting whether a file will be affected by a certain source code change (ssc), showing that

Neural Networks can predict classes of fine-grained source code changes [34]. Similar to this, each of the 48 low-level *change types* extracted with CHANGEDISTILLER can be used as a feature to analyze how these *change types* vary between subsequent patches. By obtaining low-level features about how the code is changed between different patches, patterns and anti-patterns can emerge, making future detection of issues possible.

4.3.2 Feature Extraction

In the following, we present a possible feature extraction process in order to capture the presented metrics of a given patch in code review with the intent to use this information in future work to predict and detect needed code changes in newly submitted patches.

Formal Representation. In order to utilize machine learning algorithms in a future prototype, the status of patches needs to be modeled in a more formal way with the metrics defined in the previous section. Initially, a dataset of code review patches needs to be curated that holds relevant information for the extraction process such as the url to fetch the patches from, dates, file paths, commit messages, removed and added line counts as well as a label stating what *change type* is needed in that particular patch. Then, the extracted features need to be transformed into a numerical matrix M, in which each column represents the status of a general j$-th$ patch in a given CR commit, and each row contains the values of each CR metric defined to model the patch status at that commit. Thus, each entry $\mathbf{M}_{[i,j]}$ of the matrix represents the value of the i$-th$ metric of the j$-th$ patch. This is needed, as machine learning models operate on numerical values only.

Feature extraction process. Figure 4.3 describes the high-level structure of a possible feature extraction process. In a first step, the dataset containing information about relevant patches is loaded. Then, for each data point the version of the file in the current patch is checked out in *repository 1* and at the same time the previous version of file is checked out in *repository 2*. This allows for computing metrics on both versions of a file and creating a diff (compare differences) of metrics to track how different characteristics of the file and patch change. In this sense, all specified metrics described in the previous section are computed for both files. For every metric, a dedicated script is run. This allows for further extensions with additional metrics in the future. In a final step, the results of each metric are written into a separate file as a numerical matrix. The computed values for each feature are then combined into a final numerical matrix for further processing.

4.3.3 Change Type Prediction

Prediction of needed changes in CRAM. Before the classification step, the columns of the matrix M need to be ordered by considering the timestamp of each CR commit. Thus, to predict the CR *change types* required on a general j$-th$ patch (i.e., a patch at a given CR commit), a sub-matrix of the dataset if considered as training set, $\mathbf{M}_{training,j}$, obtained by selecting from the original matrix M the columns associated to the CR commits (or column) occurred before the j$-th$ patch (i.e., all columns ids with $id = k$ such that $1 <= k <= j - th$).

Eventually, CR *change types* of a general j$-th$ patch can automatically be predicted by relying on the general $\mathbf{M}_{training,j}$, i.e., obtained by selecting from the original matrix M the columns with $id = k$ such that $1 <= k <= j - th$. To achieve this goal, we plan to experiment with different machine learning techniques, namely, the standard probabilistic Naive Bayes classifier, the sequential minimal optimization (SMO) algorithm[6], and the J48 tree model that have been suc-

[6]https://goo.gl/BwMjzS

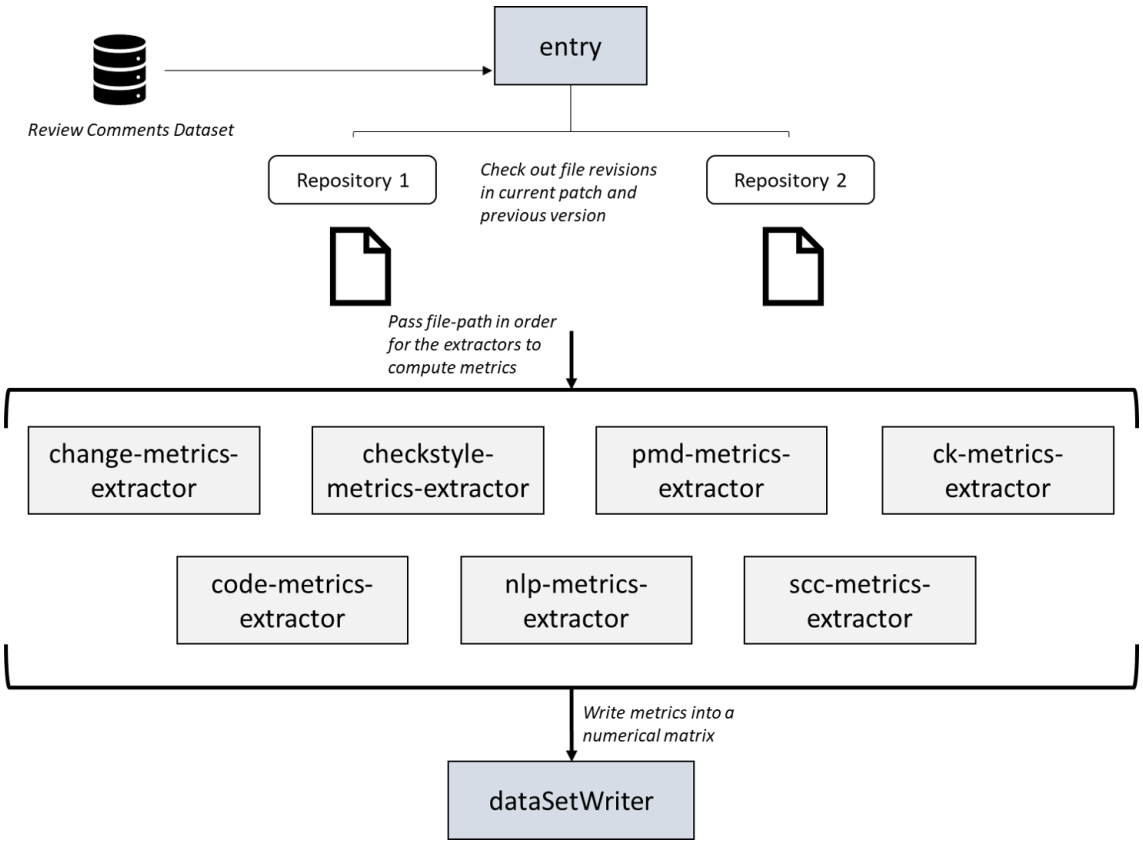

Figure 4.3: Feature Extraction Process

cessfully used for bug reports classification [5,76]. For the implementation of this approach, the Weka tool [7] offers good integration.

One important aspect that needs to be mentioned is the large number of categories in CRAM (Tables 4.4, 4.5 and 4.6). As an implementation towards the automation of MCR should be able to detect multiple defects in a submitted patch instead of only stating that it requires further changes, we require an approach that is able to do multi-label classification. To achieve this, we will use a one-vs-all strategy, a typical method for solving such problems, where one classifier is built and trained for each separate category that needs to be predicted. By running the classifier sequentially on $\mathbf{M}_{training,j}$ for every category, a label is given to each data point stating if that particular data point contains said category; explained in simpler terms, for every category, the classifier outputs a binary classification stating if the data point belong to this category or not. For the training process, the whole dataset should be divided into an 80% train and 20% test set and used for each classifier. By combining all individual classifiers, multi-label classification on a newly submitted patch can be achieved. This will allow us to (i) predict what *change types* are further needed in this patch and to (ii) support developers in proposing solutions to the respective *change type* in that category.

[7] https://www.cs.waikato.ac.nz/ml/weka/

Validation. To evaluate the performances of the prediction, well-known information retrieval metrics, such as precision, recall, and F-measure [7] will be used.

Precision (P) is the measure of the result relevancy. It is defined as the number of true positives (tp) in relation to false positives (fp) plus true positives (tp), where the value 1.0 represents a precision of 100%.

$$P = \frac{tp}{(tp+fp)}$$

Recall (R) is the measurement of how many relevant results are returned. It is defined as the number of true positives (tp) in relation to the number of false negatives (fn) plus true positives (tp), where the value 1.0 represents a recall of 100%.

$$R = \frac{tp}{(tp+fn)}$$

F1-Measure (F1) is a score that considers both the precision and recall and is represented by its general definition as the harmonic mean between precision and recall.

$$F1 = 2 * \frac{P*R}{P+R}$$

Summary. In this *proof of concept*, we showed how various metrics proposed by the developers in our study can be extracted in a future implementation of a tool. The future implementation will be able to compare how metrics change over the course of reviews as well as analyze and learn which characteristics are predictors for issues, as well as detect recurrent problems in MCR. Furthermore, we highlighted the usefulness of these metrics for particular categories in CRAM (code metrics, change metrics, CK metrics, ASAT warnings, commit messages, CHANGEDISTILLER changetypes). More specifically, metric features such as static code metrics were mostly mentioned in relation to Functionality *change types*, whereas NLP features obtained from commit messages and the code should prove more relevant when considering categories such as Documentation and Style.
By following this approach, it would be possible to predict on a file-level the different categories of CRAM in which further changes are needed. This represents a first step towards the partly or full automation of MCR. In future work, specific approaches need to be implemented into tools in order to detect and fix the more fine-grained issues within each predicted category, such as (i) proposing auto-fixes; (ii) auto-highlighting of issues and explanation of the problem. Furthermore, to efficiently support developers in MCR, it is crucial that these tools are not only developed, but are integrated into their reviewing processes.
We believe that by considering the findings in this thesis and following the approach proposed in our *proof of concept*, a useful tool can be implemented that is able to facilitate the reviewing activities of developers and reduce the review effort considerably.

4.4 Threats to Validity

Threats to construct validity. Threats to *construct validity* concern the design of our study. We advertised the survey through social media channels and by opportunistic sampling, and thus we could not avoid the lack of conscientious responses. Also, given the evaluation of the survey, some responses included imprecisions, in fact, some answers given were superficial or incomplete. In order to mitigate these threats, ambiguous and incomplete answers were discarded during the evaluation of the survey. Another threat to construct validity are the steps involved in the development of `CRAM`, as this involved manual classification of code review changes and the qualitative analysis of the feedback gathered in the survey. Indeed, there is a level of subjectivity involved when deciding if a feedback or review change belongs to a certain category. To alleviate some of these threats we based `CRAM` on three different sources of change type information: (i) manual classification of ten different Java open source projects, where each assignment was double-checked by another author; (ii) integration of an existing taxonomy from literature; and (iii) the evaluation and classification of the feedback from developers, which was again double-checked by another author.

Threats to internal validity. Threats to *internal validity* concern factors that could have influenced the results of our study. A primary threat exists concerning the definition of our taxonomy, as some categories of review changes could be missing or even overlap with others. To mitigate this threat, we grouped the taxonomy into high and low level categories in order to minimize the risk of an incomplete taxonomy.

Threats to external validity. Threats to *external validity* concern the generalization of our findings. Indeed, our investigation of review changes is limited to ten Java open source projects, all within the Eclipse ecosystem. We alleviated some of these threats by choosing projects with different domains and sizes, but evidently this does not allow to draw conclusions for MCR in open source software in general. Furthermore, the dataset we studied was limited, consisting of less than 700 review comments obtained from Gerrit, which might restrict the generalizability of our findings in settings such as other programming languages, projects and review tools. Nonetheless, participants in our study have various backgrounds and most of them stated they had more than 8 years of programming experience, which clearly does not limit the findings of this study to only the Java open source environment.

Chapter 5

Conclusion

This thesis empirically investigated approaches and tools that, from a *developer point of view*, are still needed to facilitate MCR activities. In a first step we elicited a taxonomy (CRAM) characterizing the most critical and recurrent *change types* in MCR by: (i) quantitatively and qualitatively analyzing code review changes in ten Java open source projects; (ii) validating our taxonomy with an existing classification from literature [15] and (iii) conducting a survey with 52 developers to find missing *change types* in our taxonomy and to investigate current developer's automation needs regarding review changes and activities. Table 5.1 presents our main findings for each research questions and states future work that is still needed.

Our elicited taxonomy CRAM captures code review *change types* that are not considered in current taxonomies. We found that *change types* regarding (i) ***licensing*** and ***security issues***; (ii) badly composed strings and wrong or missing imports; (ii) potential typos in either in-line comments or Javadocs; (iii) the removal of commented out code; (iv) the application of bad testing practices; and finally, (iv) the handling of architectural changes to the system are critical to developers and should be considered in MCR taxonomies. Additionally, we found in our investigation that around 80% of *change types* occurring in MCR are related to maintainability issues, supporting previous findings of studies in MCR [15, 44]. Furthermore, our investigations show the surfacing of various changes in CR, such as changes in ***non-source-code artifacts***, due to the availability of emerging technologies (e.g., cloud based technologies) and practices (e.g., Continuous Delivery and Continuous Integration). More specifically, developers perform additional changes in continuous delivery and integration configuration files, changes in files for runtime configuration, changes in static analysis tools configuration files and other non-source-code artifacts (e.g., runtime configurations, commits and external software documentations). Moreover, the evaluation of our survey revealed a gap between the expectation of developers towards the code review and the actual outcome of reviews. As mentioned above, due to emerging technologies and practices such as CI and CD as well as new cloud-based technologies, developers expect more exhaustive feedback in MCR and believe MCR processes should adapt to these new practices and technologies, which further widens the gap between expectations and actual outcomes of code review. We found that from a developers' point of view, the following automation would support developers in their reviewing activities and help towards closing the existing gap: (i) solutions for consistent naming within a project; (ii) refactoring recommendations; (iii) detection and fixing of critical security and performance issues; (iv) and tools that recommend the right configurations for CD/CI pipelines. Our investigations into how developers believe these solutions could be automated found that the general approach consists of the following steps: (i) conduct a manual analysis to investigate patterns, anti-patterns, change metrics and documentation metrics; (ii) leverage NLP or machine learning techniques in combination with static code analysis; and then (iii) characterize and predict future changes or detect issues in a submitted patch. Furthermore, it is particularly important for developers that potential solutions are tailored to the specific team

Table 5.1: Main Findings and Needed Future Work

Research Question	Main Findings	Needed Future Work
RQ_1	**Emerging practices** (CI/CD) and technologies (cloud-based) push developers to perform additional changes and tasks in MCR.	Investigation is needed in how these emerging practices and technologies shape the activities of developers in MCR and how current review tools accommodate these new requirements.
RQ_1	Additional *change types* in code review are related to **non-source-code artifacts** and concern the configuration in CI/CD files, changes in runtime configurations and ASATs configurations.	Investigation is needed in the correct configurations of CD/CI and ASATs, specifically, how software teams define policies and standards. Our research highlighted that many issues in MCR are related to bad configurations. By reducing the number of wrong configurations, the review effort can be reduced.
RQ_2	Developers require specific automation for Documentation (**consistent naming and automatic generation of required documentation**), Style (**evaluation of style consistency with the existing code base**), Structure (**refactoring suggestions and detection of unused/deprecated/dead code**), Functionality (**security, resource consumption and performance**) and support for emerging practices and technologies (**recommend and improve CD/CI and ASAT configurations**).	In our study, we surveyed 52 developers coming from different background. While they provided interesting and valuable insights and recommendation towards the automation of MCR, further research in different settings is needed to capture developer needs to the full extent.
RQ_3	To achieve automation in MCR, the most promising approach is to (i) perform a manual analysis to investigate **patterns/anti-patterns**, **change metrics** and **documentation metrics**; (ii) then leverage **NLP** or **machine learning** techniques in combination with **static code analysis**; (iii) to characterize and **predict further changes** or detect issues in a submitted patch.	Developers provided insights into how the automation of these solutions could be achieved. While we received valuable feedback, which we discussed in this thesis, more research is needed how each specific solution can be implemented and reliably used in MCR.
RQ_3	Different metrics are relevant for modeling a given patch in code review. Metrics such as **static code metrics** are most useful in relation to Functionality *change types*, whereas **NLP features** obtained from commit messages and the code is more relevant when considering categories such as Documentation and Style. Furthermore, **change metrics**, **CK-metrics**, **ASATs warnings** and **low-level source code changes** should be analyzed to characterize a given patch in MCR in order to automatically detect and predict needed code changes.	While we believe that the metrics proposed in the *proof of concept* are suited to model a given patch and capture much of its features, future work should concentrate on additional features to conceive more fine-grained details about a patch. For instance, our proof of concept does not yet consider the team composition or expertise of different developers.

RQ_1: *What types of changes occur during code reviews?*

RQ_2: *What are the emerging automation needs of developers in MCR?*

RQ_3: *What approaches are feasible for the automation of MCR?*

and project they are used in. For instance, tools should consider and learn the team's naming policies before suggesting naming-fixes.

Based on our findings regarding the automation needs of developers, we proposed a proof of concept to show how to extract various metrics characterizing a given patch in code review. In the future, we plan to investigate the automation of certain aspects of MCR by considering the insights found in our analysis, by implementing a prototype that leverages Machine Learning techniques and NLP to predict *change types* in code review patches.

We hope that this study was able to provide valuable insights into recurrent *change types* in CR, developers expectations towards MCR, automation needs and possible solutions toward the automation of MCR by investigating some of the most critical and emerging issues developers have to deal with in MCR, and to propose ways how developers' reviewing activities can be facilitated by novel tools and approaches.

Bibliography

[1] CheckStyle. `http://checkstyle.sourceforge.net`. Accessed: 2018-09-03.

[2] Gerrit. `https://code.google.com/p/gerrit/`. Accessed: 2018-09-03.

[3] PMD. `http://pmd.sourceforge.net`. Accessed: 2018-09-03.

[4] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski. Software inspections: An effective verification process. *IEEE Software*, 6(3):31–36, May 1989.

[5] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement?: A text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, pages 23:304–23:318, New York, NY, USA, 2008. ACM.

[6] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 712–721, 2013.

[7] R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[8] V. Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 931–940, 2013.

[9] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 134–144, 2015.

[10] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Eng.*, 22(10):751–761, 1996.

[11] T. Baum, K. Schneider, and A. Bacchelli. On the optimal order of reading source code changes for review. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 329–340, 2017.

[12] G. Bavota and B. Russo. Four eyes are better than two: On the impact of code reviews on software quality. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pages 81–90, 2015.

[13] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. The secret life of patches: A firefox case study. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 447–455, 2012.

[14] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey. Investigating technical and non-technical factors influencing modern code review. *Empirical Software Engineering*, 21(3):932–959, 2016.

[15] M. Beller, A. Bacchelli, A. Zaidman, and E. Jürgens. Modern code reviews in open-source projects: which problems do they fix? In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 202–211, 2014.

[16] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 470–481. IEEE Computer Society, 2016.

[17] B. W. Boehm and V. R. Basili. Software defect reduction top 10 list. *IEEE Computer*, 34(1):135–137, 2001.

[18] A. Bosu and J. C. Carver. Impact of peer code review on peer impression formation: A survey. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 133–142, Oct 2013.

[19] A. Bosu, J. C. Carver, C. Bird, J. D. Orbeck, and C. Chockley. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. *IEEE Trans. Software Eng.*, 43(1):56–75, 2017.

[20] A. Bosu, M. Greiler, and C. Bird. Characteristics of useful code reviews: An empirical study at microsoft. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*, pages 146–156, 2015.

[21] R. Chatley and L. Jones. Diggit: Automated code review via software repository mining. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 567–571, 2018.

[22] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[23] J. Czerwonka and M. Greiler. Code reviews do not find bugs. how the current code review best practice slows us down. IEEE – Institute of Electrical and Electronics Engineers, May 2015.

[24] M. Di Penta, L. Cerulo, and L. Aversano. The life and death of statically detected vulnerabilities: An empirical study. *Information & Software Technology*, 51(10):1469–1484, 2009.

[25] A. Di Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall. What would users change in my app? summarizing app reviews for recommending software changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 499–510, New York, NY, USA, 2016. ACM.

[26] P. Duvall, S. M. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[27] P. M. Duvall. Continuous integration. patterns and antipatterns. *DZone refcard #84*, 2010.

[28] V. Efstathiou and D. Spinellis. Code review comments: language matters. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 69–72, 2018.

[29] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems*, 38(2-3):258–287, June 1999.

[30] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

[31] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, 2007.

[32] D. M. Germán and Y. Manabe. Ninka, a license identification tool for source code.

[33] D. M. Germán, G. Robles, G. Poo-Caamaño, X. Yang, H. Iida, and K. Inoue. "was my contribution fairly reviewed?": a framework to study the perception of fairness in modern code reviews. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 523–534, 2018.

[34] E. Giger, M. Pinzger, and H. C. Gall. Can we predict types of code changes? an empirical analysis. In *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*, pages 217–226, 2012.

[35] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, pages 35–44, 2010.

[36] C. Hannebauer, M. Patalas, S. Stünkel, and V. Gruhn. Automatically recommending code reviewers based on their expertise: an empirical comparison. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 99–110, 2016.

[37] M. Höst and C. Johansson. Evaluation of code review methods through interviews and experimentation. *Journal of Systems and Software*, 52(2-3):113–120, 2000.

[38] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[39] C. F. Kemerer and M. C. Paulk. The impact of design and code reviews on software quality: An empirical study based on PSP data. *IEEE Trans. Software Eng.*, 35(4):534–550, 2009.

[40] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan. What do mobile app users complain about? *IEEE Software*, 32(3):70–77, 2015.

[41] S. Kim and M. D. Ernst. Which warnings should I fix first? In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 45–54, 2007.

[42] O. Kononenko, O. Baysal, and M. W. Godfrey. Code review quality: how developers see it. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 1028–1038, 2016.

[43] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey. Investigating code review quality: Do people and participation matter? In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pages 111–120, 2015.

[44] M. V. Mantyla and C. Lassenius. What types of defects are really discovered in code reviews? *IEEE Transactions on Software Engineering (TSE)*, 35(3):430–448, 2009.

[45] T. J. McCabe. A complexity measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[46] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The impact of code review coverage and code review participation on software quality: a case study of the qt, vtk, and ITK projects. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 192–201, 2014.

[47] M. Menarini, Y. Yan, and W. G. Griswold. Semantics-assisted code review: an efficient toolchain and a user study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 554–565, 2017.

[48] L. Moreno and A. Marcus. Automatic software summarization: the state of the art. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 530–531, 2018.

[49] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 181–190, New York, NY, USA, 2008. ACM.

[50] V. Nguyen, S. Deeds-Rubin, T. Tan, and B. Böhm. A sloc counting standard. 2007.

[51] M. Nurolahzade, S. M. Nasehi, S. H. Khandkar, and S. Rawal. The role of patch review in software evolution: An analysis of the mozilla firefox. In *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, pages 9–18, 2009.

[52] A. Ouni, R. G. Kula, and K. Inoue. Search-based peer reviewers recommendation in modern code review. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*, pages 367–377, 2016.

[53] M. Paixão, J. Krinke, D. Han, and M. Harman. CROP: linking code reviews to source code changes. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, pages 46–49, 2018.

[54] S. Panichella. Summarization techniques for code, change, testing, and user feedback (invited paper). In *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests, VST@SANER 2018, Campobasso, Italy, March 20, 2018*, pages 1–5, 2018.

[55] S. Panichella, V. Arnaoudova, M. D. Penta, and G. Antoniol. Would static analysis tools help developers with code reviews? In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 161–170, 2015.

[56] S. Panichella, A. D. Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, pages 281–290, 2015.

[57] D. L. Parnas and M. Lawford. The role of inspection in software quality assurance. 29(8):674–676, 2003.

[58] M. M. Rahman, C. K. Roy, and R. G. Kula. Predicting usefulness of code review comments using textual features and developer experience. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 215–226, 2017.

[59] G. Rajesh Choudhary, S. Kumar, K. Kumar, A. Mishra, and C. Catal. Empirical analysis of change metrics for software fault prediction. 67, 03 2018.

[60] P. C. Rigby. *Understanding Open Source Software Peer Review: Review Processes, Parameters and Statistical Models, and Underlying Behaviours and Mechanisms*. PhD thesis, University of Victoria, BC, Canada, 2011.

[61] P. C. Rigby and D. M. German. A preliminary examination of code review processes in open source projects. Technical Report DCS-305-IR, University of Victoria, January 2006.

[62] P. C. Rigby, D. M. German, and M. D. Storey. Open source software peer review practices: a case study of the apache server. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 541–550, 2008.

[63] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm. Continuous deployment at facebook and OANDA. In *Companion proceedings of the 38th International Conference on Software Engineering (ICSE Companion)*, pages 21–30.

[64] D. Spadini, M. F. Aniche, M. D. Storey, M. Bruntink, and A. Bacchelli. When testing meets code review: why and how developers review tests. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 677–687, 2018.

[65] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K. Matsumoto. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 141–150, 2015.

[66] F. Thung, Lucia, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 50–59, 2012.

[67] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall. Context is king: The developer perspective on the usage of static analysis tools. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 38–49, 2018.

[68] C. Vendome, D. M. Germán, M. D. Penta, G. Bavota, M. L. Vásquez, and D. Poshyvanyk. To distribute or not to distribute?: why licensing bugs matter. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 268–279, 2018.

[69] H. R. Wagner. The discovery of grounded theory: Strategies for qualitative research. *Social Forces*, 46(4):555, 1968.

[70] S. Wagner, J. Jurjens, C. Koller, and P. Trischberger. Comparing bug finding tools with reviews and tests. In *Proceedings of the 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems*, pages 40–55, 2005.

[71] P. Weißgerber, D. Neu, and S. Diehl. Small patches get in! In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pages 67–76, 2008.

[72] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 334–344. IEEE Press, 2017.

[73] M. B. Zanjani, H. H. Kagdi, and C. Bird. Automatically recommending peer reviewers in modern code review. *IEEE Trans. Software Eng.*, 42(6):530–543, 2016.

[74] T. Zhang, M. Song, J. Pinedo, and M. Kim. Interactive code review for systematic changes. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 111–122, 2015.

[75] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. C. Gall. Analyzing apis documentation and code to detect directive defects. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 27–37, 2017.

[76] Y. Zhou, Y. Tong, R. Gu, and H. C. Gall. Combining text mining and data mining for bug report classification. *Journal of Software: Evolution and Process*, 28(3):150–176, 2016.

[77] T. Zimmermann. Changes and bugs - mining and predicting development activities. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*, pages 443–446, 2009.